OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT -I

Object Orientation – System development – Review of objects - inheritance - Object

relationship – Dynamic binding – OOSD life cycle – Process – Analysis – Design – prototyping – Implementation – Testing- Overview of Methodologies

UNIT -II

Rambaugh methodology, OMT – Booch methodology, Jacobson methodology – patterns – Unified approach – UML – Class diagram – Dynamic modeling.

UNIT –III

Introduction - UML – Meta model - Analysis and design - more information. Outline

Development Process: Overview of the process-Inception - Elaboration-construction- refactoring-patternstransmission-iterative development -use cases.

UNIT -IV

OO Design axioms – Class visibility – refining attributes – Methods –Access layer – OODBMS – Table – class mapping view layer

UNIT –V

Interaction diagram-package diagram-state diagram-activity diagram-deployment

diagram - UML and programming

TEXT BOOK

1. Ali Bahrami, “Object Oriented System Development”, McGraw-Hill International

Edition 2017.

REFERENCES

1. Booch G., “Object oriented analysis and design”, Addison- Wesley Publishing

Company 3 rd edition.

2. Rambaugh J, Blaha.M. Premeriani, W., Eddy F and Loresen W., “ObjectOriented

Modeling and Design”, PHI

3. Martin Fowler, Kendall Scott, "UML Distilled", Addision Wesley

4. Eriksson, "UML Tool Kit", Addison Wesley.

COURSE PLAN

Select a course (a theory subject): **Object oriented Analysis and Design**

Select a unit: **Unit I**

Select a Topic: **INTRODUCTION**

Object Orientation – System development – Review of objects - inheritance - Object relationship – Dynamic binding – OOSD life cycle – Process – Analysis – Design – prototyping – Implementation – Testing- Overview of Methodologies

1. Objectives

- To learn the basis of OO Analysis and design
- To have clear idea about traditional and modern SW development Methodologies and OOPS concepts.
- To identify objects, relationships, services and attributes
- To introduce the concept of Object-oriented system development lifecycle
- Discuss the overview of Object oriented methodologies

2. Outcomes

On completion of this course the students will be able to

- Explain and implement the SW development Methodologies.
- Explain and apply basic OOPS concepts

3. Pre-requisites

1. Software Engineering
2. OOPS

Plan for the lecture delivery

1. **How to plan for delivery – black board / ppt / animated ppt / (decide which is good for this topic)**

   Blackboard/Power Point Presentation

2. **How to explain the definition and terms with in it**
   - What is an object?
   - What is OOA, OOD, and OOP?
   - What is meant by object orientation?
   - What are the basic concepts of oops?
   - Give an example for an object, object properties and methods
   - What is meant by inheritance and its types
   - What are the steps followed by system development?

3. **How to start the need for any derivation / procedure / experiment / case study**

   How to develop the object oriented software system

   The Object-Oriented approach of Building Systems takes the objects as the basis. For this, first the system to be developed is observed and analyzed and the requirements are defined as in any other method of system development. Once this is often done, the objects in the required system are identified.

   For example, in the case of a Banking System, a customer is an object, a chequebook is an object, and even an account is an object.

4. **Units and their physical meaning to make them understand practical reality and comparison between different units**

   Two Approaches, for developing software system

   a. **Traditional Approach**

   b. **Objected-Oriented Approach**

Traditional Approach	**Object oriented Approach**

Collection of procedures	Combination of data and functions
Focuses on function and procedures different styles and methodologies for each step of process	Focuses on object,classes,modules that can be easily replaced,modifies and reused
Moving from one phase to another phase is complex	Moving from one phase to another phase is easier
Increases complexity	Reduce complexity and redundancy
Increases duration of the project	Decreases duration of the project

❖ It adapts to

- Changing requirements
- Easier to maintain
- More robust
- Promote greater design
- Code reuse

5. What is the final conclusion?

Learn the object oriented concepts, relationship between objects and understand how to develop the object oriented software system

6. **How to put it in a nut shell**

- OO Methods enables to develop set of objects that work together to build software
- Logical and physical models and iterative and incremental development concepts in OOA and OOD Need for the Unified Modeling Language (UML) and model-driven development

7.Important points for understanding / memorizing / make it long lasting

Object-oriented analysis and design (OOAD) is a software engineering approach

- **Analysis** — understanding, finding and describing concepts in the problem domain.
- **Design** — understanding and defining software solution/objects that represent the analysis concepts and will eventually be implemented in code.

- System development activities consist of system analysis, modeling, design, implementation, testing and maintenance.

- Two Approaches for software development
    - Traditional Approach
    - Objected-Oriented Approach
- BENEFITS OF OBJECT ORIENTATION
    - Faster development,
    - Reusability,
    - Reduced complexity
    - Increased quality
- Each object has SBI(**S**tate,**B**ehaviour,**I**dentity)
- OO Relationship
    - Generalization (parent-child relationship)
    - Association (student enrolls in course)
    - Aggregation(whole/part of relationship)
    - Composition(strong form of aggregation)
- The **unified approach (UA)** is a methodology for software development.
- **Booch, Rumbaugh, Jacobson methodologies** gives the best practices, processes and guidelines for OO oriented software development.

8. Questions and cross questions beyond the topic.

- Differentiate object oriented and object based technology.
- How does object-oriented development eliminate duplication
- Differentiate correspondence and correctness measures
- Why is reusability important? How does object oriented software development promote reusability?

9. Final conclusion

- Learn the object oriented concepts, relationship between objects and understand how to develop the object oriented software system
- The advocates of OOSD claim many advantages including easier modeling, increased code reuse, higher system quality and easier maintenance.
- It is well understood that analysis and design are extremely critical aspects of successful systems development especially in the case of OOSD.

- As the development of any successful information system must begin with a well-conceived and implemented analysis and design.

Introduction

What is Object-Oriented Analysis and Design

OOA: we find and describe business objects or concepts in the problem domain

OOD: we define how these software objects collaborate to meet the requirements. Attributes and methods.

OOP: Implementation: we implement the design objects in, say, Java, C++, C#, etc.

Definition

Object-oriented analysis and design (OOAD) is a popular technical approach for analyzing, designing an application, system, or business by applying the object oriented paradigm and visual modeling throughout the development life cycles for better communication and product quality**.**

Software development

Software development is dynamic and always undergoing major change.

Systems development refers to all activities that go into producing an information systems solution. Systems development activities consist of systems analysis, modeling, design, implementation, testing and maintenance.

A **software development methodology** is a series of processes describe how the work is to be carried out to achieve the original goal based on the system requirements.

TWO ORTHOGONAL VIEWS OF THE SOFTWARE

- Two Approaches,
  - Traditional Approach
  - Objected-Oriented Approach

Traditional Approach

Here Algorithms + Data structures = Programs. "A software system is a set of mechanisms for performing certain actions on certain data."

Difference between Traditional and Object Oriented Approach

Traditional Approach	Object oriented Approach
Collection of procedures	Combination of data and functions
Focuses on function and procedures different styles and methodologies for each step of process	Focuses on object,classes,modules that can be easily replaced, modifies and reused
Moving from one phase to another phase is complex	Moving from one phase to another phase is easier
Increases complexity	Reduce complexity and redundancy
Increases duration of the project	Decreases duration of the project

Object- oriented systems Development methodology:-

Object-Oriented development offers a different model from the traditional software development approach, which is based on functions and procedures.

In an object-Oriented system, everything is an object and each object is responsible for itself. A windows object is responsible for things like opening, sizing, and closing itself.

Why an Object Orientation?

The systems are easier to adapt to changing requirements, easier to maintain, more robust, and promote greater design and code reuse. Object- oriented development allows us to create module of functionality.

Here or some reasons why object- Orientation works:

- *Higher level of abstraction.* The top-down approach supports abstraction at the function level. The Object-oriented approach supports abstraction at the object level. Since objects encapsulate both data (attributes) and functions (methods), they work at a higher level of abstraction.
- *Seamless transition among different phases of software development.* The traditional approach to software development requires different styles and methodologies for each sep of the process.
- *Encouragement of good programming techniques.* A class in an object- oriented system carefully delineates between its interface (specifications of what the class can do) and the implementations of that interface (how the class does what it does).

- *Promotion of reusability.* Objects are reusable because they are modeled directly out of a real-world problem domain. Each object stands by itself or within a small circle of peers(other objects).

Object Basics:

Objects:

The term **Object** means a combination of data and logic that represents some real world entity.

When developing an object – oriented application, two basic questions always arise:

- What objects does the application need?
- What functionality should those objects have?

OBJECTS ARE GROUPED IN CLASSES

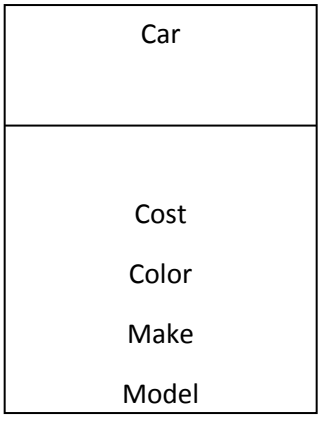
A class is a specification of structures (instance variables), behavior (methods), and inheritance for objects.

Classes are important mechanisms for classifying objects. In an object- oriented system, a *method* or behavior of an object is defined by its class. Each object is an instance of a class. There may be many different classes.

Object state and properties (attributes):-

Properties represent the state of an object.

Example: the properties of a car, such as color, manufacturer, and cost, are abstract descriptions.



The attributes of a Car object.

For color, we could choose to use a sequence of characters such as red, etc.

Behavior (Object Behavior and methods);-

We can drive a car, we can ride an elephant, or the elephant can eat a peanut. Each of these statements is a description of the object's behavior. In the object model, object behavior is described in methods or procedures. A method implements the behavior of an object.

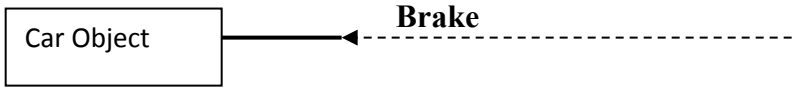
Messages (Objects respond to messages):

An object's capabilities are determined by the methods defined for it. Methods conceptually are equivalent to the functions definitions used in procedural languages. Objects perform operations in response to messages. For example, when you press on the brake pedal of a car, you send a stop message to the car object.

Messages essentially are nonspecific function calls: we would send a draw message to a chart when we want the chart to draw itself. A message is different from a subroutine call, since different objects can respond to the same message in different ways. For example, cars, motorcycles, and bicycles will all respond to a stop message, but the actual operations performed are object specific.

Methods are similar to functions, procedures, or subroutine in more traditional programming languages, such as COBAL, Basic, or C. The area where methods and functions differ, however, is in how they are invoked.

Figure: Objects respond to messages according to methods defined in its class.



Information Hiding (and Encapsulations):-

Information hiding is the principle of concealing the internal data and procedures of an object and providing an interface to each object in such a way to reveal as little as possible about its inner workings.

On object is said to encapsulate the data and a program. This means that user cannot see the inside of the object "capsule," but can use the object by calling the object's methods.

Encapsulation or information hiding is a design goal of an object-oriented system. Rather than allowing an object direct access to another object's data, a message is sent to the target object requesting information.

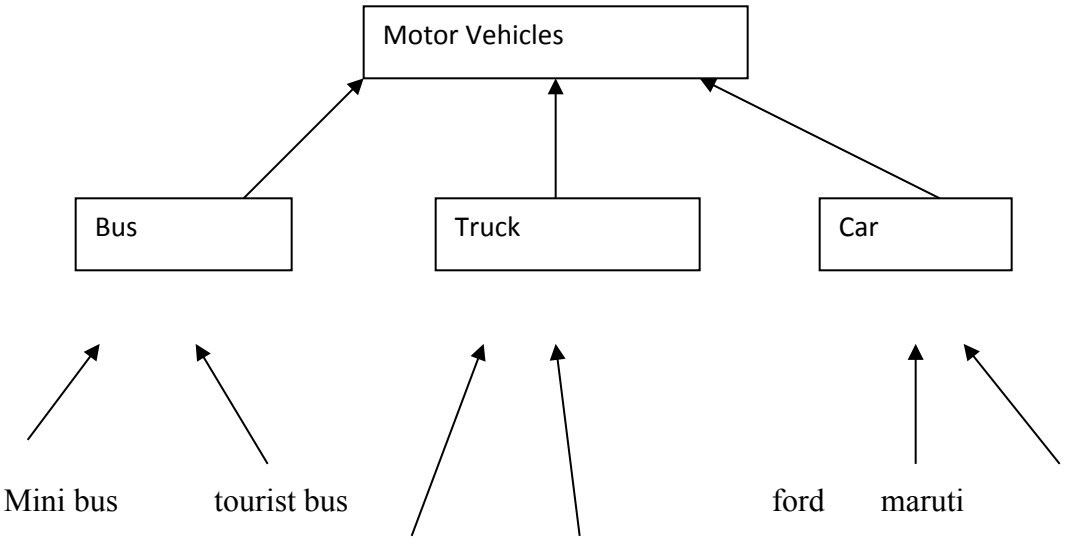
Another issue is per-object or per-class protection. In **per-class protection,** the most common form (e.g., Ada, C++, Eiffel), class methods can access any object of that class and not just the receiver. In **per-object protection**, methods can access only the receiver.

A car engine is an example of encapsulation.

CLASS HIERARCHY:-

An object-oriented system organizes classes into a subclass- superclass hierarchy. Different behaviors are used as the basics for making distinctions between classes and subclasses. At the top of the **Class hierarchy** are the most general classes and at the bottom are the most specific.

A subclass inherits all of the properties and methods defined in its **superclass.**



A car class defines how a car behaves. The ford class defines the behavior of ford class.

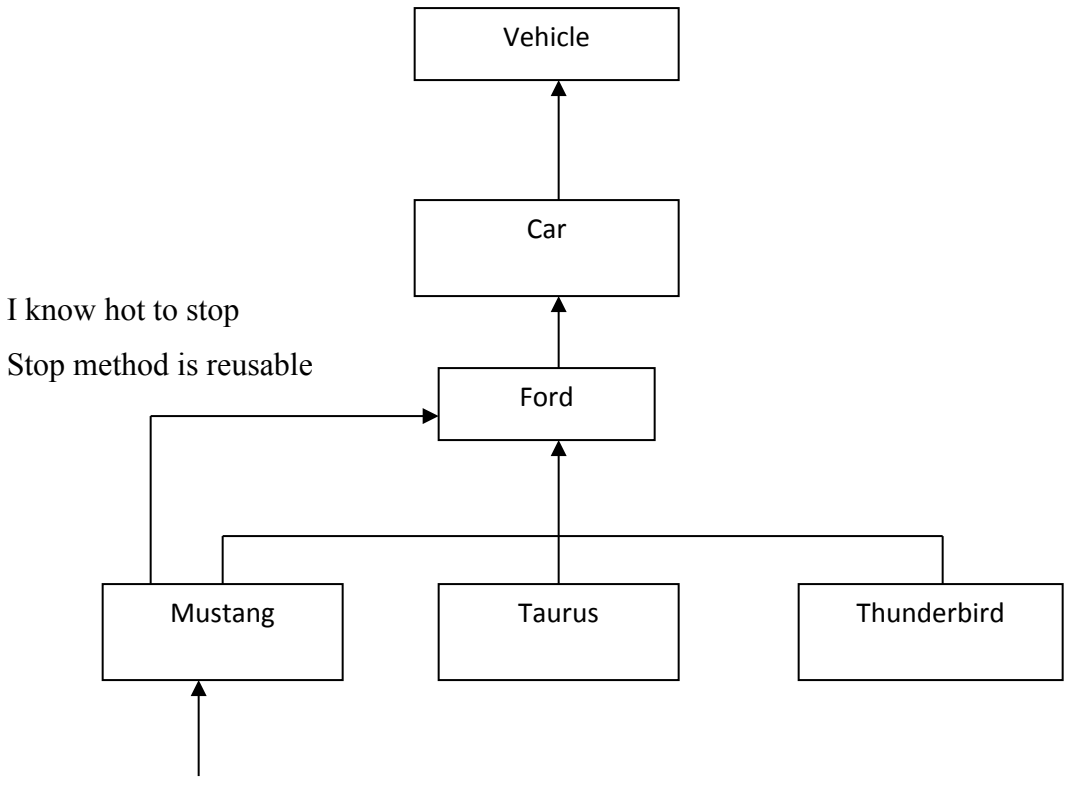
Formal or abstract classes have no instances but define the common behaviors that can be inherited by more specific classes.

In some object- oriented languages, the terms *superclass* and *subclasses* are used instead of **base** and **derived**.

Inheritance:-

Inheritance is the property of object- oriented systems that allows objects to be built from other objects. Inheritance allows explicitly taking advantage of the commonality of objects when constructing new classes. Inheritance is a relationship between classes where one class is the parent class of another (derived) class. The parent class is also known as the *base class* or *superclass.*

For example the car class defines the general behavior of cars. The ford class inherits the general behavior from the car class and adds behavior specific to fords.



Dynamic inheritance allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, changing base classes changes the properties and attributes of a class.

More specifically, *dynamic inheritance* refers to the ability to add, delete, or change parents form objects (or classes) at run time.

Multiple inheritance:-

Some object- oriented systems permit a class to inherit its state (attributes) and behaviors from more than one super class. This kind of inheritance is referred to as **Multiple Inheritance.**

POLYMORPHISM:-

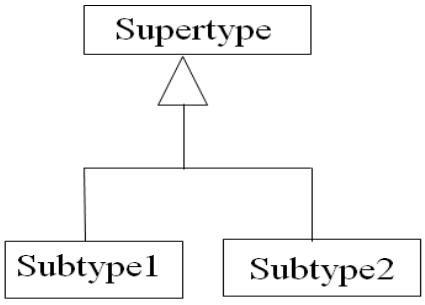
Poly means "many" and *morph* means "form". In the context of object- oriented systems, it means objects that can take on or assume many different forms. **Polymorphism** means that the same operation may behave differently on different classes.

Booch [1-3] defines *polymorphism* as the relationship of objects of many different classes by some common superclass.

Object Relationships

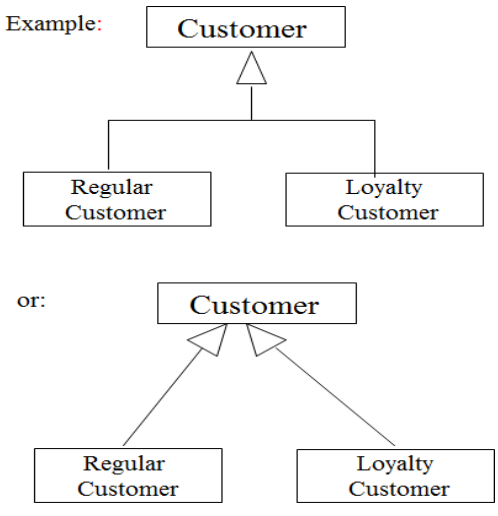
- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association (link b/w objects/class)
- Associations can be further classified as
  - Aggregation(whole/part of relationship)
  - Composition(strong form of aggregation)

OO Relationships: Generalization

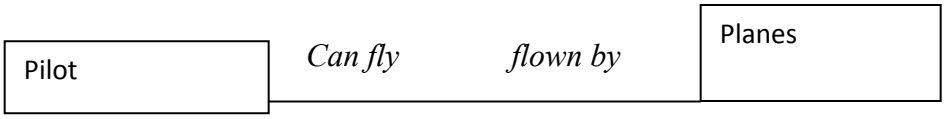


- Generalization expresses a parent/child relationship among related classes.

- Used for abstracting details in several layers



Associations represent the relationship between objects and classes. For example, in the statement "a pilot *can fly planes"*(below figure), the italicized term is an association.



Associations are bidirectional; that's means they can be traversed in both directions, perhaps with different connotations. The direction implied by the name is the forward directions; the opposite direction is the inverse direction. For example, *can fly* connects a pilot to certain airplanes. The inverse of *can fly* could be called is flown by.

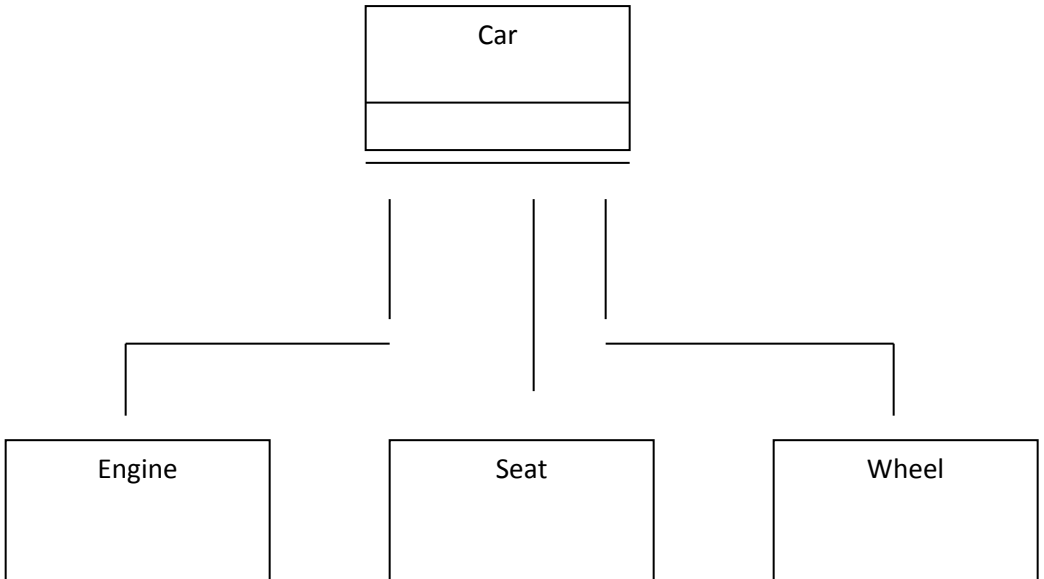
An important issue in association is **cardinality,** which specifies how many instances of one class may relate to a single instance of an associated class.

Consumer- Producer Association

A special form of association is a consumer-producer relationship, also known as a **client-server association** or a **use relationship.** The **consumer-producer relationship** can be viewed as one-way intersection: One object requests the service of another object.

Aggregations (Aggregation and object containment):-

Each object has an identity, one object can refer to other objects. This is known as ***aggregation,*** where an attribute can be an object itself. For instance, a car object is an aggregation of engine, seat, wheels, and other objects.



Identity (Object and identity):-

A special feature of object-oriented system is that every object has its own unique and immutable identity. An object's identity comes into being when the object is created and continues to represent that object from then on. This identity never is confused with another object, even if the original object has been deleted.

In an object system, object identity often is implemented through some kind of **object identifier** (OID) or unique identifier (UID).

Dynamic Binding (Static and Dynamic Binding):-

The process of determining (dynamically) at run time which functions to invoke is termed **dynamic binding.**

Making this determination earlier, at compile time, is called **static binding.**

Static binding optimizes the calls; dynamic binding occurs when polymorphic calls are issued.

Dynamic binding allows some methods invocation decisions to be deferred until the information is known.

Persistence (Object Persistence):-

Objects have a lifetime. They are explicitly created and can exist for a period of time that, traditionally, has been the duration of the process in which they were created. A filer or a database can provide support for objects having a longer lifeline longer than the duration of the process for which they were created.

From a language perspective, this characteristic is called **object persistence.**

An object can persist beyond application session boundaries, during which the object is stored in a file or a database, in some file or database form. The object can be retrieved in another application session and will have the same state and relationship to other objects as at the time it was saved.

The lifetime of an object can be explicitly terminated.

Meta-Classes

Is a class an object? A class is an object, a class belongs to a class called a meta-class, or a class of classes.

For example, classes must be implemented in some way; perhaps with dictionaries for methods, instances, and parents and methods to perform all the work of being a class. This can be declared in a class named **meta-class.**

The meta-class also can provide services to application programs, such as returning a set of all methods, instances, or parents for review (or even modification)

Object – Oriented Systems Development Life Cycle:-

Introduction:

The essence of the software Development Process that consists of analysis, design, implementation, testing, and refinement is to transform users' needs into a software solution that satisfy those needs.

The Software Development Process:

System development can be viewed as a process. Within the process, it is possible to replace one sub process has the same interface as the old one to allow it to fit into the process as a whole.

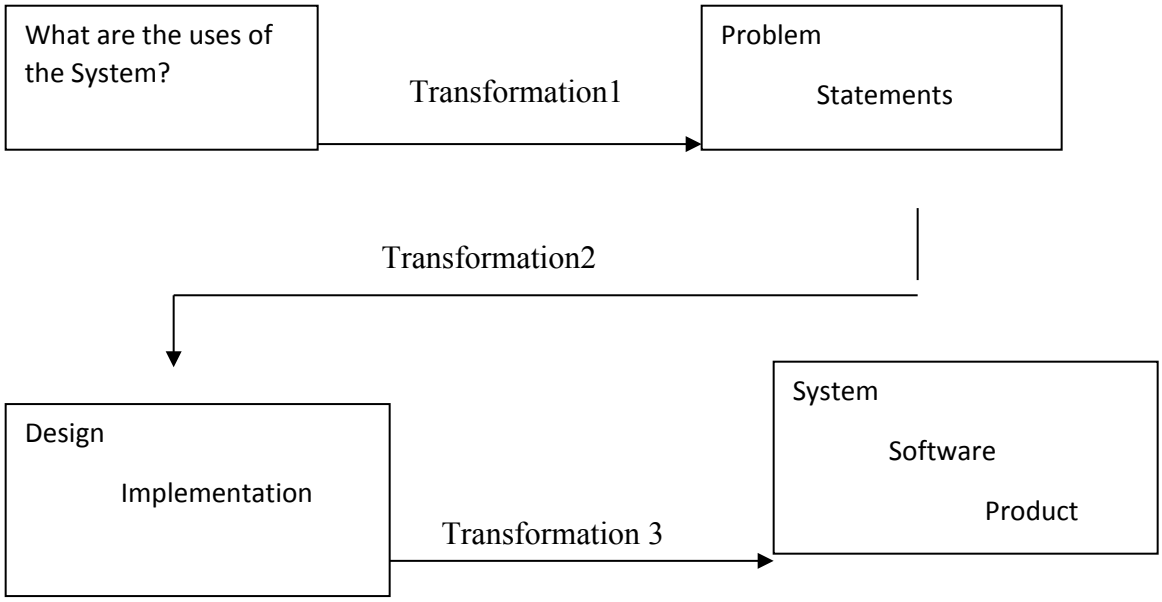
For example, Object oriented approach

The process can be divided into small, interacting phases- sub processes. The subprocesses must be defined in such a way that they are clearly spelled out; to allow each activity to be performed as independently of other subprocesses as possible. Each subprocess must have the following [1]:

- A description in terms of how it works.
- Specification of the input required for the process.
- Specification of the output to be produced.

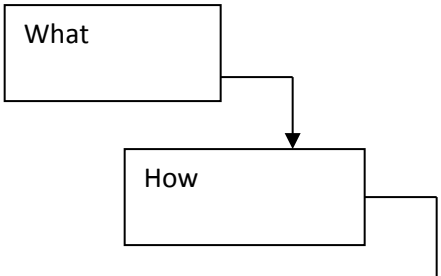
The software development process also can be divided into smaller, interacting subprocesses. Generally, the software development process can be viewed as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation.

- Transformation1 (Analysis): It translates the users need into the System requirements and responsibilities. The way they use the system can provide insight into the user's requirements.
- Tranformation2 (Design): It begins with a problem statement and ends with a detailed design that can be transformed into an OS. This transformation includes the bulk of the software development activity, including the definition of how to build the software, its development, and it's testing.



- Transformation3 (Implementation) refines the detailed design into the system deployment that will satisfy the user needs. This takes into account the equipment, procedures, peoples and the like. It represents embedding the software product within its operational environment.

An example of the software development process is the **waterfall approach**, which starts with deciding what is to be done (what is the problem).



The waterfall model is the best way to manage a project with a well understood product, especially very large products.

Building high Quality software:-

The software process transforms the users' needs via the application domain to a software solution that satisfies those needs. Once the system exists, we must test it to see if is free of bugs.

To achieve high quality in software we need to be able to answer the following questions:

- How do we determine when the system is ready for delivery?
- Is it now an operational system that satisfies users' needs?
- Is it correct and operating as we thought it should?
- Does it pass an evaluation process?

There are four quality measures: Correspondence, Correctness, Verification and validation.

Correspondence measures how well the delivered system matches the needs of the operational environment, as described in the original requirements statements.

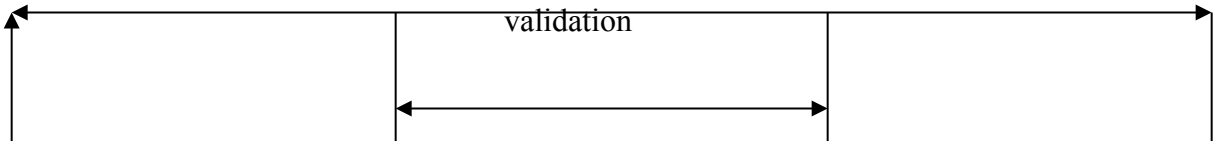
Validation is the task of predicting correspondence.

Correctness measures the consistency of the product requirements with respect to the design specification.

Verification is the exercise of determining correctness.

Verification and validation, answer the following question:

- *Verification: Am I building the product right?*
- *Validation: Am I building the right product?*



verification

correctness

correspondence

OBJECT ORIENTED SYSTEM DEVELOPMENT: A USECASE DRIVEN APPROACH:

The object-oriented **software development life cycle** (SDLC) consists of three macro processes: Object-oriented analysis, Object-oriented design, and object-oriented implementation.

Object-Oriented system development includes these activities:

- Object-oriented analysis-use case driven
- Object- Oriented design
- Prototyping
- Component- based development
- Incremental Testing

Object-Oriented Analysis--Use-case Driven:

The object-oriented Analysis phase of software development is concerned with determining the system requirements and identifying classes and their relationship to other classes in the problem domain. To understand the system requirements, we need to identify the users or the actors.

The intersection among objects' roles to achieve a given goal is called **collaboration.**

Object-Oriented Design:-

The goal of Object-Oriented design (OOD) is to design the classes identified during the analysis phase and the user interface. During this phase, we identify and define additional objects and classes that support implementation of the requirements.

First, build the object model based on objects and their relationships, then iterate and refine the model:

- Design and refine classes.
- Design and refine attributes.
- Design and refine methods.
- Design and refine Structures.
- Design and refine Associations.

Prototyping:-

Prototypes have been categorized in various ways. They are

- A **horizontal prototype** is a simulation of the interface but contains no functionality. This has the advantages of being very quick to implement, providing a good overall feel of the system.
- A **vertical prototype** is a subset of the system features with complete functionality. The principal advantage of this method is that the few implemented functions can be tested in great depth.
- An **analysis prototype** is an aid for the incremental development of the ultimate software solution.

The purpose of this review is threefold:

1. To demonstrate that the prototype has been developed according to the specification and that the final specification is appropriate.

2. To collect information about errors or other problems in the system, such as user interface problems that need to be addressed in the intermediate prototype stage.

3. To give management and everyone connected with the project the first (or it could be second or third)

Implementation: concept- Based Development

Component-based manufacturing makes products available to the marketplace that otherwise would be prohibitively expensive.
For example, computer-aided software engineering (CASE) tools.

Component-based development (CBD) is an industrialized approach to the software development process.

Rapid application development (RAD) is a set of tools and techniques that can be used to build an application faster than typically possible with traditional method. The terms often is used in conjuction with software prototyping.

Reusability:-

A major benefit of object-oriented system development is reusability, and this is the most difficult promise to deliver on. For an object to be really reusable, much more effect must be spent designing it.

Software components for reuse by asking the following questions:

- Has my problem already solved?
- Has my problem been partially solved?
- What has been done before to solve a problem similar to this one?

The reuse strategy can be based on the following:

- Information hiding (encapsulation).
- Conformance to naming standards.
- Creation and administration of an object repository.
- Encouragement by strategic management of reuse as opposed to constant development.
- Establishing targets for a percentage of the objects in the project to be reused.

Assignment questions

- Explain the relationship association, composition and aggregation with an example
- How does object oriented methodology differ from other programming methodologies? Explain with an example.
- How is software development viewed? What are the various phases of OOSD life cycle? What is waterfall approach? List out its limitations

UNIT-II

OBJECT ORIENTED Methodologies

OBJECTIVES:

At the end of this chapter, students should be able to

- Define and understand the various object-oriented
- UML – Drawing Class diagrams and the process in Dynamic modeling

COURSE OUTCOME

- Good knowledge in Patterns and Unified approach.
- Identification of objects, attributes, methods, associations will be easy in designing a class

Object-Oriented Analysis Models

Object-oriented Analysis

Object-oriented analysis starts with a traditional structured specification, and adds the following information:

- A list of all objects - A list describing the data contents of each *noun*, or physical entities in the DFD.
- A list all system behaviors - A list of all *verbs* within the process names such as *Prepare order* summary report, *generate* invoices, etc.
- A list of the associate the primary behaviors (services) with each object - Each object will have behaviors which uniquely belong to the object. Other objects may request the behavior of the object.
- A description of the contracts in the system - A contract is an agreement between two objects, such that one object will invoke the services of the other.
- A behavior script for each object - A script describes each initiator, action, participant, and service.
- A classification for each object and the object relationships - Generate an entity/relationship model and a generalization hierarchy (IS-A) for each object, using traditional E/R or normalization techniques.

Over the past 12 years there have numerous books about different approaches to object analysis bit they all contain these common elements. Now that we see the basic analysis requirements, let's explore the basic methodologies for object-oriented analysis.

Different Models for Object Analysis

Unlike the traditional systems analysis where user requirements are gathered and then specifications are put on the requirements and users are then asked to sign off on the specifications, the object methodologies use a more iterative process where the requirements and specifications are reviewed repeatedly and the users are heavily involved.

Object technology has many different methodologies to help analyze and design computer systems. We will review four of the more popular systems: Rumbaugh, Booch, Coad-Yourdon, and Shlaer-Mellor. In most cases these methodologies are very similar, but each has its own way

to graphically represent the entities. To understand and use these four methodologies would become difficult, if not impossible, for all projects. If need be, it is possible to use concepts from one method with concepts from another technique, basically creating your own object development technique. The most important point is to remember is that the final outcome is what really matters, not the choice of one analysis technique over another technique. Remember, it is more important to do proper analysis and design to meet user requirements than it is to just follow a blind, meaningless procedure.

The traditional systems development approach is sometimes referred to as the waterfall method. By waterfall, object analyst's follow a logical progression through analysis, design, coding, testing, and maintenance. Unfortunately system development seldom fits this kind of structured approach. End-users are notorious for changing their minds or identifying some feature that they forgot to identify. These changes in requirements can happen at any phase of system development and the analyst must struggle to accommodate these changes into the system. What it means to the systems analyst is that you have to go back to whatever step in the development life cycle and make the necessary changes that will then cascade these changes through the entire system. For example, suppose that our end-users are in the testing phase when they realize that they need an additional screen. This would require a change to the initial requirements document, which would, in turn, cascade to analysis, design, and so on.

The object-oriented methodologies require a more iterative process with the same five steps. The iterative process either adds new or more clearly defines existing properties, unlike the traditional approach that would re-hash specifications that are already done. The iterative process helps to reduce confusion around what the system is really suppose to do and what the users really want. The object-oriented software development methods make the assumption that user requirements will change. However it doesn't matter which programming language you use, be it FORTRAN or C++. Furthermore, it doesn't matter which system development technique you use, you will follow the same five steps in system development. It is just, how these five steps are applied that will make the difference in your system development project.

The Rumbaugh method

The Rumbaugh method is listed first because it is these authors favorite, and we find it a very friendly and easy methodology. For traditional system analyst's, the Rumbaugh's methodology is the closest to the traditional approach to system analysis and design, and beginners will recognize

familiar symbols and techniques. The Rumbaugh methodology has its primary strength in object analysis but it also does an excellent job with object design. Rumbaugh has three deliverables to the object analysis phase; the Object model, the Dynamic model, and the functional model. These three models are similar to traditional system analysis, with the additions for the object model, including definitions of classes along with the classes variables and behaviors. The Rumbaugh object model is very much like an entity relationship diagram except that there are now behaviors in the diagram and class hierarchies. The dynamic model is a "state transition" diagram that shows how an entity changes from one state to another state. The functional model is the equivalent of the familiar data flow diagrams from a traditional systems analysis.

The Booch method

Booch's methodology has its primary strength in the object system design. Grady Booch has included in his methodology a requirements analysis that is similar to a traditional requirements analysis, as well as a domain analysis phase. Booch's object system design method has four parts, the logical structure design where the class hierarchies are defined, the physical structure diagram where the object methods are described. In addition, Booch defines the dynamics of classes in a fashion very similar to the Rumbaugh method, as well as an analysis of the dynamics of object instances, where he describes how an object may change state.

The Coad-Yourdon method

Coad-Yourdon methodology has its primary strength in system analysis. Their methodology is based on a technique called "SOSAS", which stands for the five steps that help make up the analysis part of their methodology. The first step in system analysis is called "Subjects", which are basically data flow diagrams for objects. The second step is called "Objects", where they identify the object classes and the class hierarchies. The third step is called "Structures", where they decompose structures into two types, classification structures and composition structures. Classification structures handle the inheritance connection between related classes, while composition structures handle all of the other connections among classes. The next step in analysis is called "Attributes", and the final step is called "Services", where all of the behaviors or methods for each class are identified.

Following analysis, Coad and Yourdon define four parts that make up the design part of their methodology. The steps of system design are:

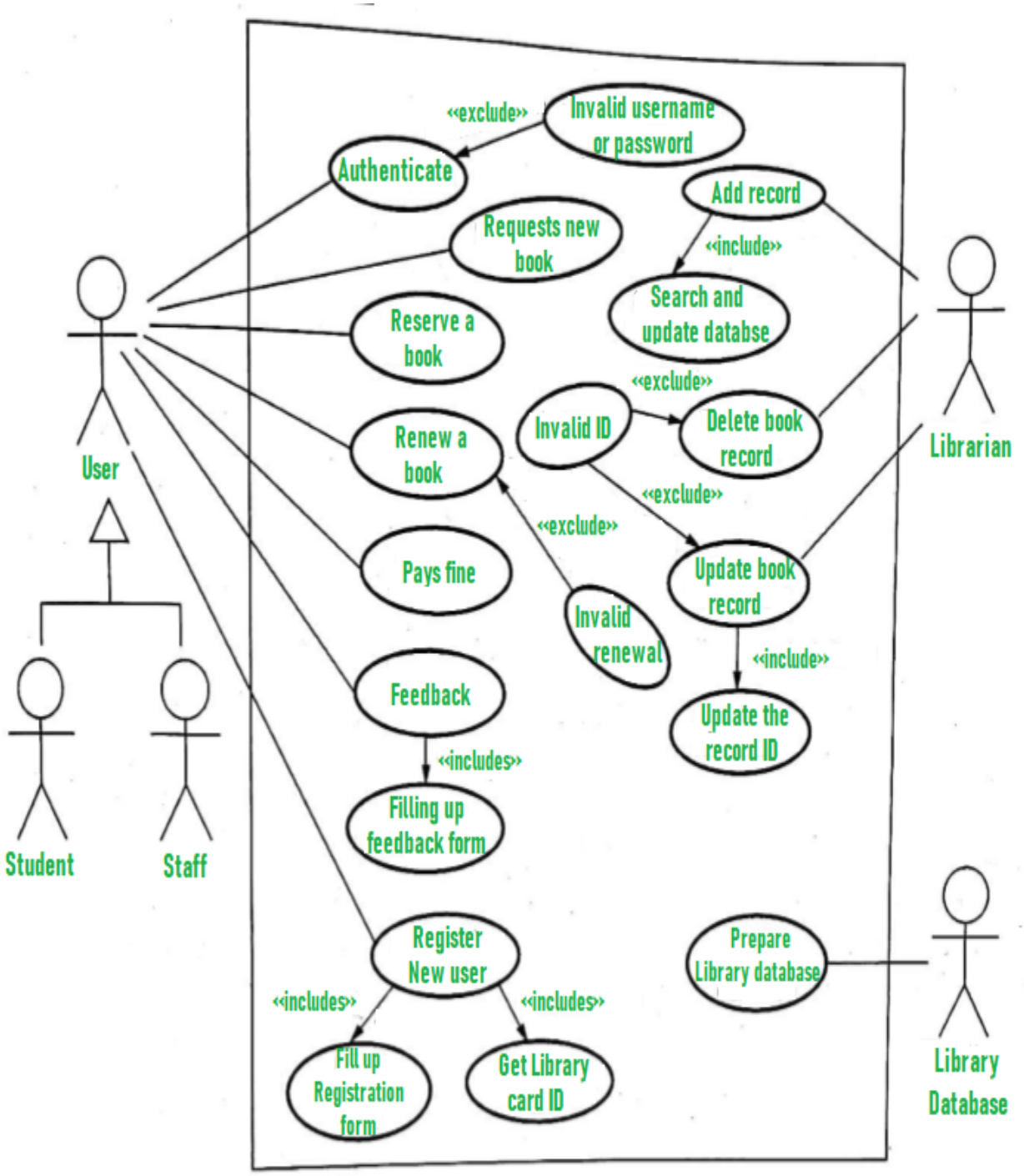
- The problem domain component - This will define the classes that should be in the problem domain.
- The human interaction component - These steps defines the interface classes between objects.
- The task management component - This is where system-wide management classes are identified.
- The data management component - This design step identifies the classes needed for database access methods.

The Shlaer-Mellor method

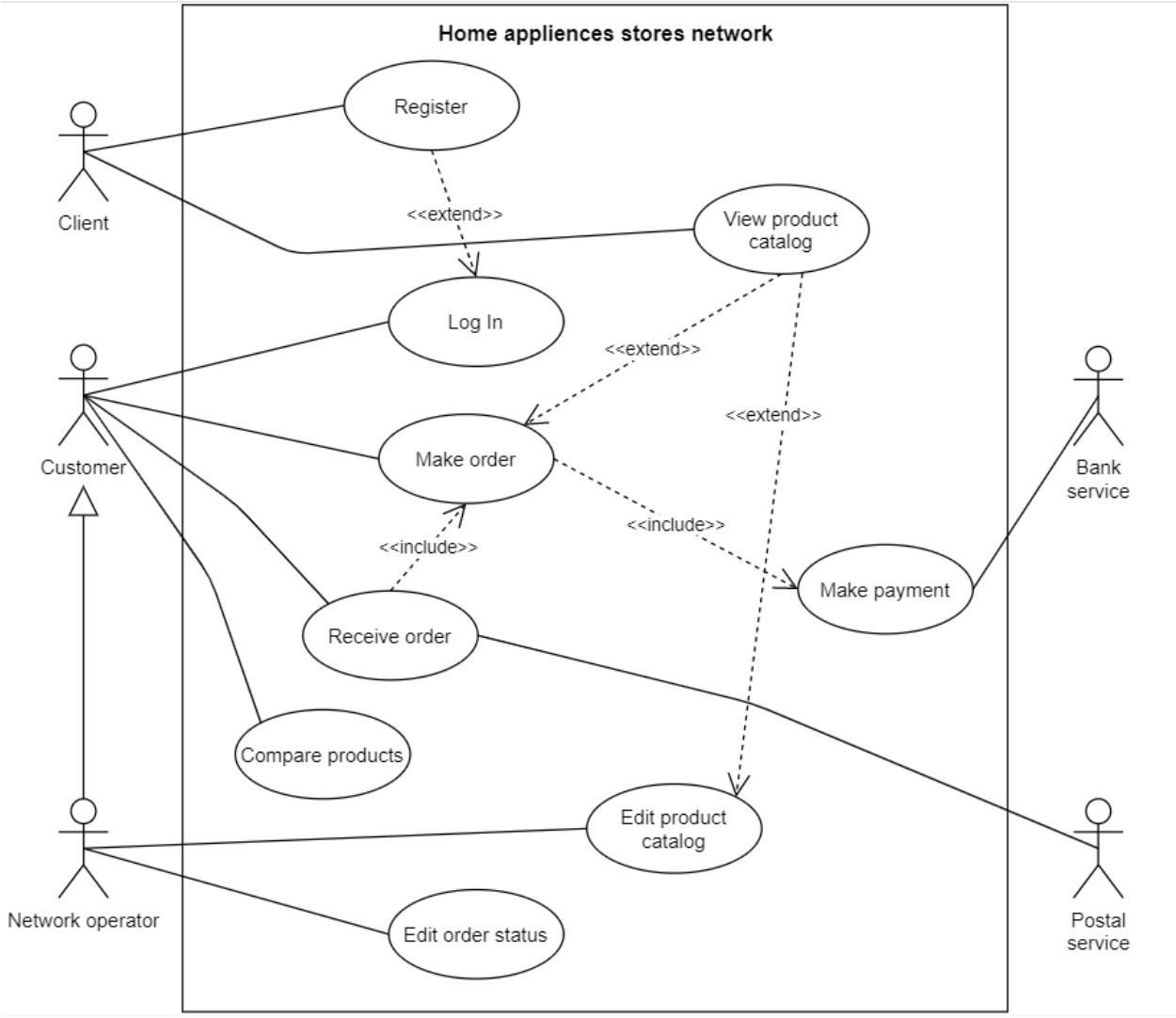
Shlaer-Mellor methodology has its primary strength in system design and is considered somewhat weak on analysis. The Shlaer-Mellor methodology includes three models; the information model, the state model, and the process model. The information model contains objects, variables, and all the relationships between the objects, and is basically a data model for the system. The state model records the different states of objects and changes that can occur between the objects. The process model is really not much more than a traditional data flow diagram.

Now that we have covered the basics of the object approach, let's take a look at how a real-world object is created by using these techniques.

UML Diagram -Library Management system


«exclude»
Invalid username or password
Authenticate
Add record
Requests new book
«include»
Search and update databse
Reserve a book
«exclude»
Invalid ID
Delete book record
Renew a book
«exclude»
«exclude»
Pays fine
Update book record
Invalid renewal
«include»
Feedback
Update the record ID
«includes»
Filling up feedback form
Register New user
Prepare Library database
«includes»
«includes»
Fill up Registration form
Get Library card ID
User
Librarian
Student
Staff
Library Database

UML Diagram -Online Order



- It is a new system development approach, encouraging and facilitating re-use of software components.
- It employs international standard Unified Modeling Language (UML) from the Object Management Group (OMG).
- Using this methodology, a system can be developed on a component basis, which enables the effective re-use of existing components, it facilitates the sharing of its other system components.

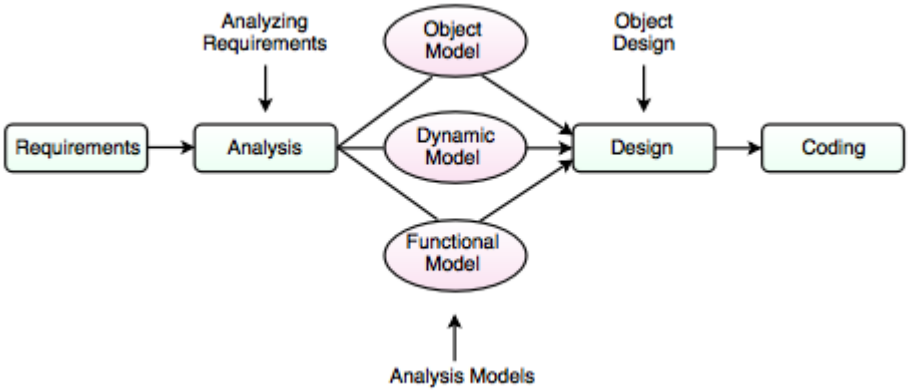
- Object Oriented Methodology asks the analyst to determine what the objects of the system are?, What responsibilities and relationships an object has to do with the other objects? and How they behave over time?

There are three types of Object Oriented Methodologies

1. Object Modeling Techniques (OMT)
2. Object Process Methodology (OPM)
3. Rational Unified Process (RUP)

1. Object Modeling Techniques (OMT)

- It was one of the first object oriented methodologies and was introduced by Rumbaugh in 1991.
- OMT uses three different models that are combined in a way that is analogous to the older structured methodologies.



a. Analysis

- The main goal of the analysis is to build models of the world.
- The requirements of the users, developers and managers provide the information needed to develop the initial problem statement.

b. OMT Models

I. Object Model

- It depicts the object classes and their relationships as a class diagram, which represents the static structure of the system.
- It observes all the objects as static and does not pay any attention to their dynamic nature.

II. Dynamic Model

- It captures the behavior of the system over time and the flow control and events in the Event-Trace Diagrams and State Transition Diagrams.

- It portrays the changes occurring in the states of various objects with the events that might occur in the system.

III. Functional Model

- It describes the data transformations of the system.
- It describes the flow of data and the changes that occur to the data throughout the system.

c. Design

- It specifies all of the details needed to describe how the system will be implemented.
- In this phase, the details of the system analysis and system design are implemented.
- The objects identified in the system design phase are designed.

2. Object Process Methodology (OPM)

- It is also called as second generation methodology.
- It was first introduced in 1995.
- It has only one diagram that is the Object Process Diagram (OPD) which is used for modeling the structure, function and behavior of the system.
- It has a strong emphasis on modeling but has a weaker emphasis on process.
- It consists of three main processes:

I. Initiating: It determines high level requirements, the scope of the system and the resources that will be required.

II. Developing: It involves the detailed analysis, design and implementation of the system.

III. Deploying: It introduces the system to the user and subsequent maintenance of the system.

3. Rational Unified Process (RUP)

- It was developed in Rational Corporation in 1998.
- It consists of four phases which can be broken down into iterations.
  I. Inception
  II. Elaboration
  III. Construction
  IV. Transition
- Each iteration consists of nine work areas called disciplines.
- A discipline depends on the phase in which the iteration is taking place.

- For each discipline, RUP defines a set of artefacts (work products), activities (work undertaken on the artefacts) and roles (the responsibilities of the members of the development team).

Objectives of Object Oriented Methodologies

- To encourage greater re-use.
- To produce a more detailed specification of system constraints.
- To have fewer problems with validation (Are we building the right product?).

Benefits of Object Oriented Methodologies

1. It represents the problem domain, because it is easier to produce and understand designs.

2. It allows changes more easily.

3. It provides nice structures for thinking, abstracting and leads to modular design.

4. Simplicity:

- The software object's model complexity is reduced and the program structure is very clear.

5. Reusability:

- It is a desired goal of all development process.
- It contains both data and functions which act on data.
- It makes easy to reuse the code in a new system.
- Messages provide a predefined interface to an object's data and functionality.

6. Increased Quality:

- This feature increases in quality is largely a by-product of this program reuse.

7. Maintainable:

- The OOP method makes code more maintainable.
- The objects can be maintained separately, making locating and fixing problems easier.

8. Scalable:

- The object oriented applications are more scalable than structured approach.
- It makes easy to replace the old and aging code with faster algorithms and newer technology.

9. Modularity:

- The OOD systems are easier to modify.
- It can be altered in fundamental ways without ever breaking up since changes are neatly encapsulated.

10. Modifiability:

- It is easy to make minor changes in the data representation or the procedures in an object oriented program.

11. Client/Server Architecture:

- It involves the transmission of messages back and forth over a network.

Comparison of various OO_Methodologies

	Booch Method	***Rumbaugh Method***	***Jacobson Method***
Approach:	Object centered approach.	Object centered approach.	User centered approach.
Phases Covered:	Analysis, design and implementation phases.	Analysis, design and implementation phases.	All phases of life phase cycle.
Strength:	Strong method for producing detailed object oriented design models.	Strong method for producing object model static structure of the system.	Strong method for producing user driven requirements and object oriented analysis model.
Weakness:	Focus entirely on design and not on analysis.	Cannot fully express the requirements.	Do not treat OOP to the same level as other methods.
Uni-directional Relationship:	Uses.	Directed Association.	
Bi-directional Relationship:	Associations.	Uni-directed Associations.	Acquaintance Relationships.
Diagrams used:	Class diagram, state	Data flow diagrams,	Use case diagram.

transition diagram, state transmission
object diagram, timing diagram, class/object
diagram, Module diagram.
diagram, process
diagram.

OOAD - Dynamic Modeling

The dynamic model represents the time–dependent aspects of a system. It is concerned with the temporal changes in the states of the objects in a system. The main concepts are −

- State, which is the situation at a particular condition during the lifetime of an object.
- Transition, a change in the state
- Event, an occurrence that triggers transitions
- Action, an uninterrupted and atomic computation that occurs due to some event, and
- Concurrency of transitions.

A state machine models the behavior of an object as it passes through a number of states in its lifetime due to some events as well as the actions occurring due to the events. A state machine is graphically represented through a state transition diagram.

States and State Transitions

State

The state is an abstraction given by the values of the attributes that the object has at a particular time period. It is a situation occurring for a finite time period in the lifetime of an object, in which

it fulfils certain conditions, performs certain activities, or waits for certain events to occur. In state transition diagrams, a state is represented by rounded rectangles.

Parts of a state

- **Name** − A string differentiates one state from another. A state may not have any name.
- **Entry/Exit Actions** − It denotes the activities performed on entering and on exiting the state.
- **Internal Transitions** − The changes within a state that do not cause a change in the state.
- **Sub–states** − States within states.

Initial and Final States

The default starting state of an object is called its initial state. The final state indicates the completion of execution of the state machine. The initial and the final states are pseudo-states, and may not have the parts of a regular state except name. In state transition diagrams, the initial state is represented by a filled black circle. The final state is represented by a filled black circle encircled within another unfilled black circle.

Transition

A transition denotes a change in the state of an object. If an object is in a certain state when an event occurs, the object may perform certain activities subject to specified conditions and change the state. In this case, a state−transition is said to have occurred. The transition gives the relationship between the first state and the new state. A transition is graphically represented by a solid directed arc from the source state to the destination state.

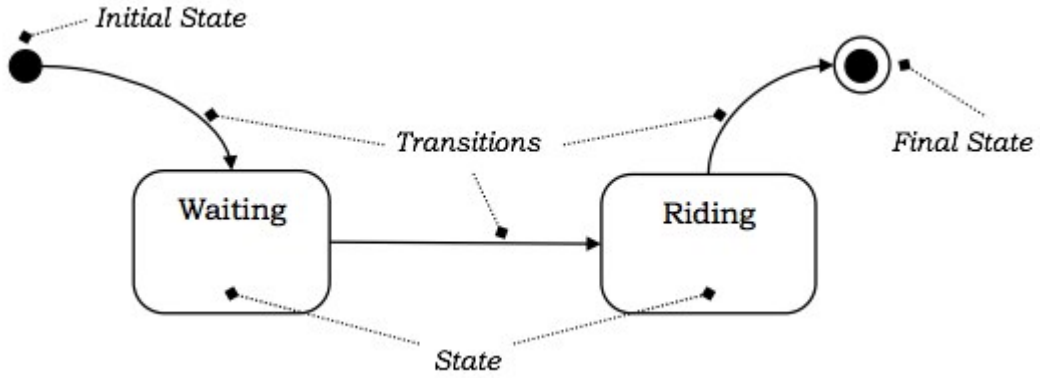
The five parts of a transition are −

- **Source State** − The state affected by the transition.
- **Event Trigger** − The occurrence due to which an object in the source state undergoes a transition if the guard condition is satisfied.
- **Guard Condition** − A Boolean expression which if True, causes a transition on receiving the event trigger.

- **Action** − An un-interruptible and atomic computation that occurs on the source object due to some event.
- **Target State** − The destination state after completion of transition.

Example

Suppose a person is taking a taxi from place X to place Y. The states of the person may be: Waiting (waiting for taxi), Riding (he has got a taxi and is travelling in it), and Reached (he has reached the destination). The following figure depicts the state transition.



Events

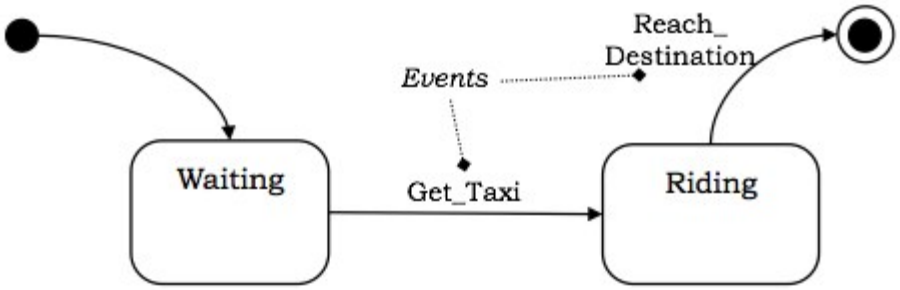
Events are some occurrences that can trigger state transition of an object or a group of objects. Events have a location in time and space but do not have a time period associated with it. Events are generally associated with some actions.

Examples of events are mouse click, key press, an interrupt, stack overflow, etc.

Events that trigger transitions are written alongside the arc of transition in state diagrams.

Example

Considering the example shown in the above figure, the transition from Waiting state to Riding state takes place when the person gets a taxi. Likewise, the final state is reached, when he reaches the destination. These two occurrences can be termed as events Get_Taxi and Reach_Destination. The following figure shows the events in a state machine.



External and Internal Events

External events are those events that pass from a user of the system to the objects within the system. For example, mouse click or key−press by the user are external events.

Internal events are those that pass from one object to another object within a system. For example, stack overflow, a divide error, etc.

Deferred Events

Deferred events are those which are not immediately handled by the object in the current state but are lined up in a queue so that they can be handled by the object in some other state at a later time.

Event Classes

Event class indicates a group of events with common structure and behavior. As with classes of objects, event classes may also be organized in a hierarchical structure. Event classes may have attributes associated with them, time being an implicit attribute. For example, we can consider the events of departure of a flight of an airline, which we can group into the following class −

Flight_Departs (Flight_No, From_City, To_City, Route)

Actions

Activity

Activity is an operation upon the states of an object that requires some time period. They are the ongoing executions within a system that can be interrupted. Activities are shown in activity diagrams that portray the flow from one activity to another.

Action

An action is an atomic operation that executes as a result of certain events. By atomic, it is meant that actions are un-interruptible, i.e., if an action starts executing, it runs into completion without being interrupted by any event. An action may operate upon an object on which an event has been triggered or on other objects that are visible to this object. A set of actions comprise an activity.

Entry and Exit Actions

Entry action is the action that is executed on entering a state, irrespective of the transition that led into it.

Likewise, the action that is executed while leaving a state, irrespective of the transition that led out of it, is called an exit action.

Scenario

Scenario is a description of a specified sequence of actions. It depicts the behavior of objects undergoing a specific action series. The primary scenarios depict the essential sequences and the secondary scenarios depict the alternative sequences.

Diagrams for Dynamic Modelling

There are two primary diagrams that are used for dynamic modelling −

Interaction Diagrams

Interaction diagrams describe the dynamic behavior among different objects. It comprises of a set of objects, their relationships, and the message that the objects send and receive. Thus, an interaction models the behavior of a group of interrelated objects. The two types of interaction diagrams are −

- **Sequence Diagram** − It represents the temporal ordering of messages in a tabular manner.
- **Collaboration Diagram** − It represents the structural organization of objects that send and receive messages through vertices and arcs.

State Transition Diagram

State transition diagrams or state machines describe the dynamic behavior of a single object. It illustrates the sequences of states that an object goes through in its lifetime, the transitions of the states, the events and conditions causing the transition and the responses due to the events.

Concurrency of Events

In a system, two types of concurrency may exist. They are −

System Concurrency

Here, concurrency is modelled in the system level. The overall system is modelled as the aggregation of state machines, where each state machine executes concurrently with others.

Concurrency within an Object

Here, an object can issue concurrent events. An object may have states that are composed of sub-states, and concurrent events may occur in each of the sub-states.

Concepts related to concurrency within an object are as follows −

Simple and Composite States

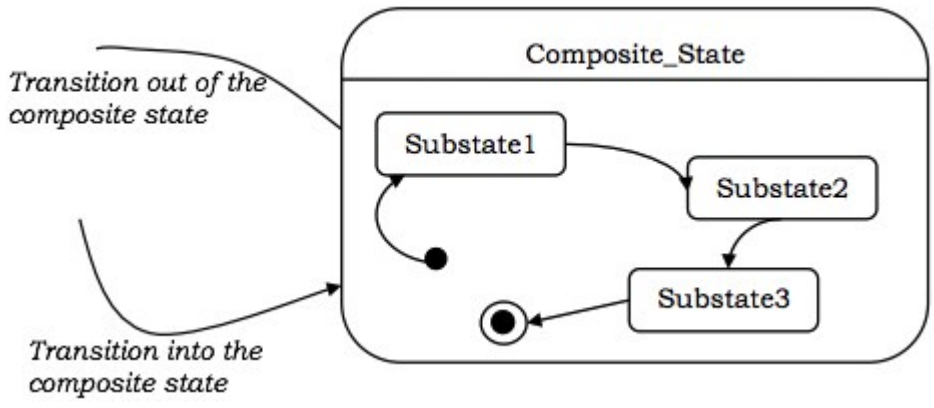
A simple state has no sub-structure. A state that has simpler states nested inside it is called a composite state. A sub-state is a state that is nested inside another state. It is generally used to reduce the complexity of a state machine. Sub-states can be nested to any number of levels.

Composite states may have either sequential sub-states or concurrent sub-states.

Sequential Sub-states

In sequential sub-states, the control of execution passes from one sub-state to another sub-state one after another in a sequential manner. There is at most one initial state and one final state in these state machines.

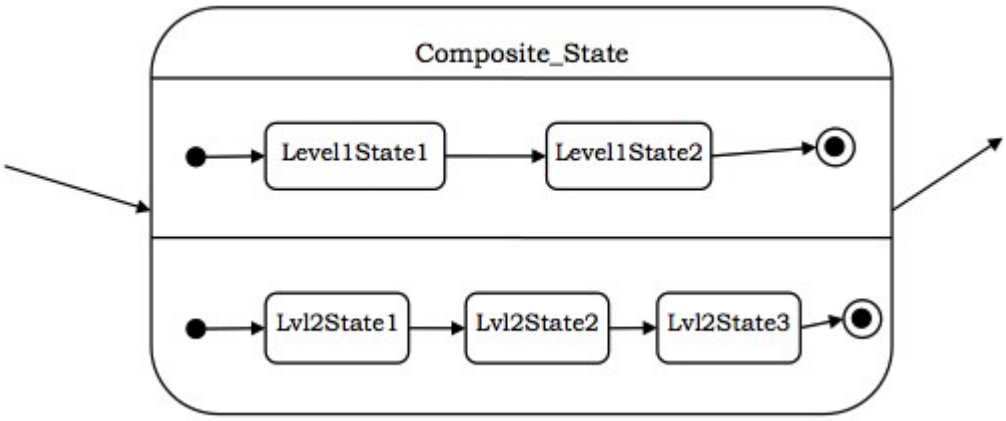
The following figure illustrates the concept of sequential sub-states.



Concurrent Sub-states

In concurrent sub-states, the sub-states execute in parallel, or in other words, each state has concurrently executing state machines within it. Each of the state machines has its own initial and final states. If one concurrent sub-state reaches its final state before the other, control waits at its final state. When all the nested state machines reach their final states, the sub-states join back to a single flow.

The following figure shows the concept of concurrent sub-states.



UML – Meta model and Outline

Development Process

OBJECTIVES:

At the end of this chapter, students should be able to

- Define and understand the Metamodel
- UML – Outline Development Process

COURSE OUTCOME

- Good Understanding in UML and metamodels.
- Different phases and processes involved in Outline Development Process

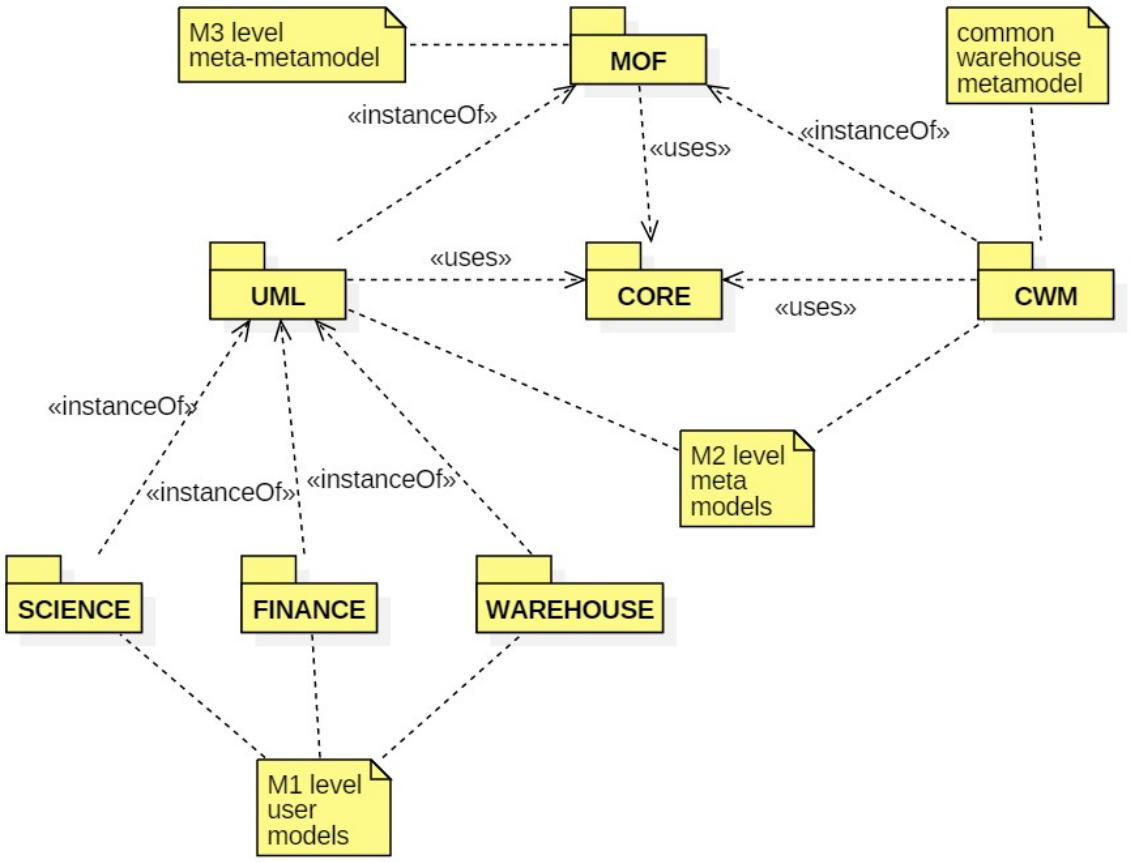
UML -Meta-Model

The UML specification [UML-OMG] is insanely complex and filled with inconsistencies. Basically, the specification uses UML to specify UML. Thus, the specification is sometimes referred to as the UML Meta-Model. I'm not sure it's worth studying the specification in too much detail unless one is building a commercial tool for processing UML diagrams.

Meta-models

The Object Management Group (OMG) is in the business of specifying standards for object-oriented developers—modeling standards (like UML), middleware standards (like CORBA), data warehousing standards (like SAS), etc.

simplified summary of the OMG specifications:



There are three levels of abstraction in the OMG specifications:

M1 = Models (i.e., models created by UML users)

M2 = Meta-Models, M1 models are instances of M2 models = {UML, CWM}

M3 = Meta-Meta-Models, M2 models are instances of M3 models = {MOF}

The Meta Object Facility (MOF) was originally a CORBA type system. It is used for defining meta-models. MOF is to domain models what EBNF is to grammars. MOF could be used to define web services as well as OO concepts.

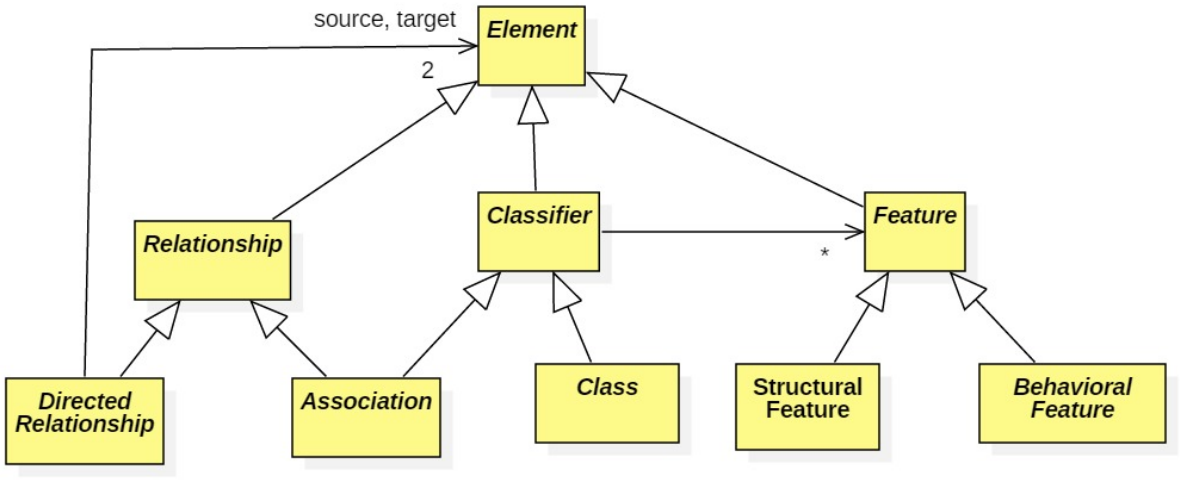
UML and CWM are M2 meta-models that instantiate MOF.

Core + UML

All of the OMG specifications depend on a tiny core of modeling concepts. [UML-KF] describes these here:

http://www.uml-diagrams.org/uml-core.html

Here's simplified version:



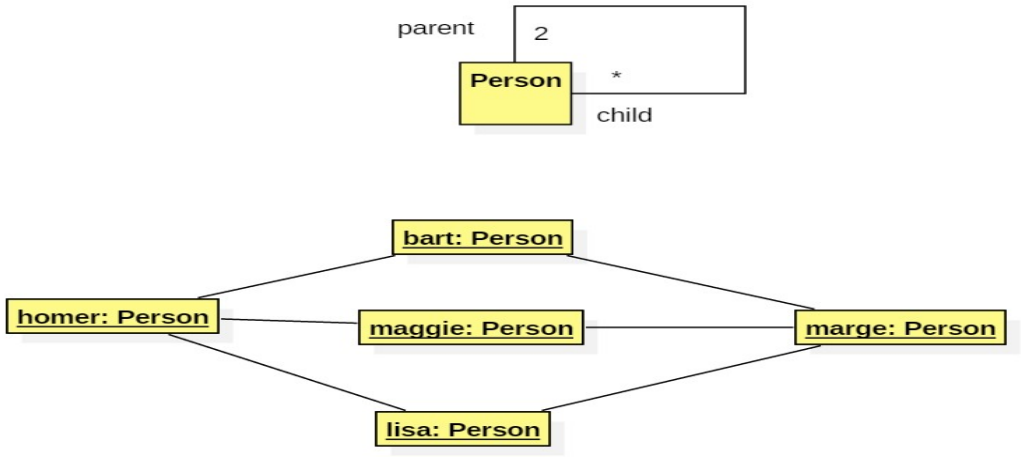
Everything that appears in a UML diagram is an element. The two most important types of elements are relationships (arrows) and classifiers (nodes).

A classifier represents a set of instances. Examples of classifiers include classes, interfaces, packages, components, use cases, actors, data types (primitive types and enumerations), associations, and collaborations. An instance of a class or interface is an object, an instance of an association is a link. A classifier can have structural and behavioral features. For a class these would correspond to attributes and operations.

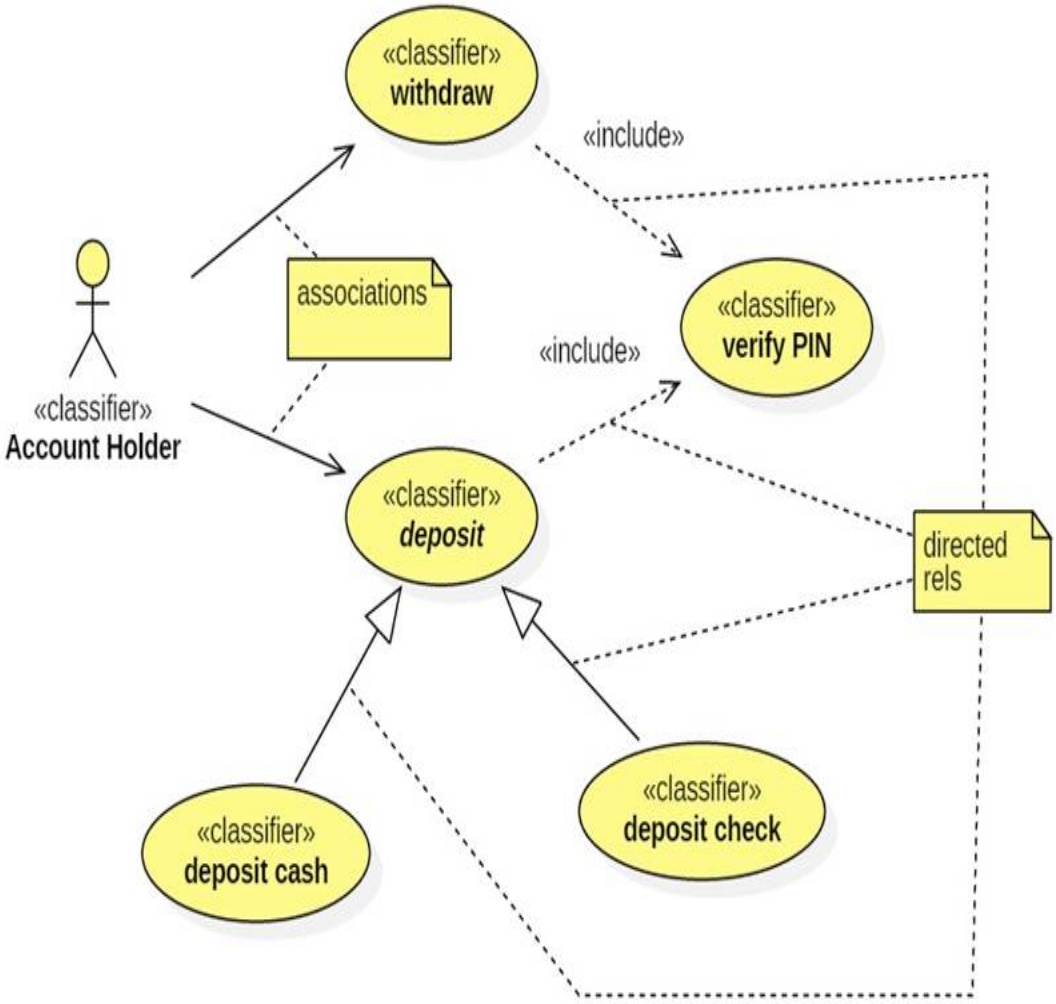
A directed relationship has a source element and a target element. Dependencies, generalizations, and realizations are types of directed relationships.

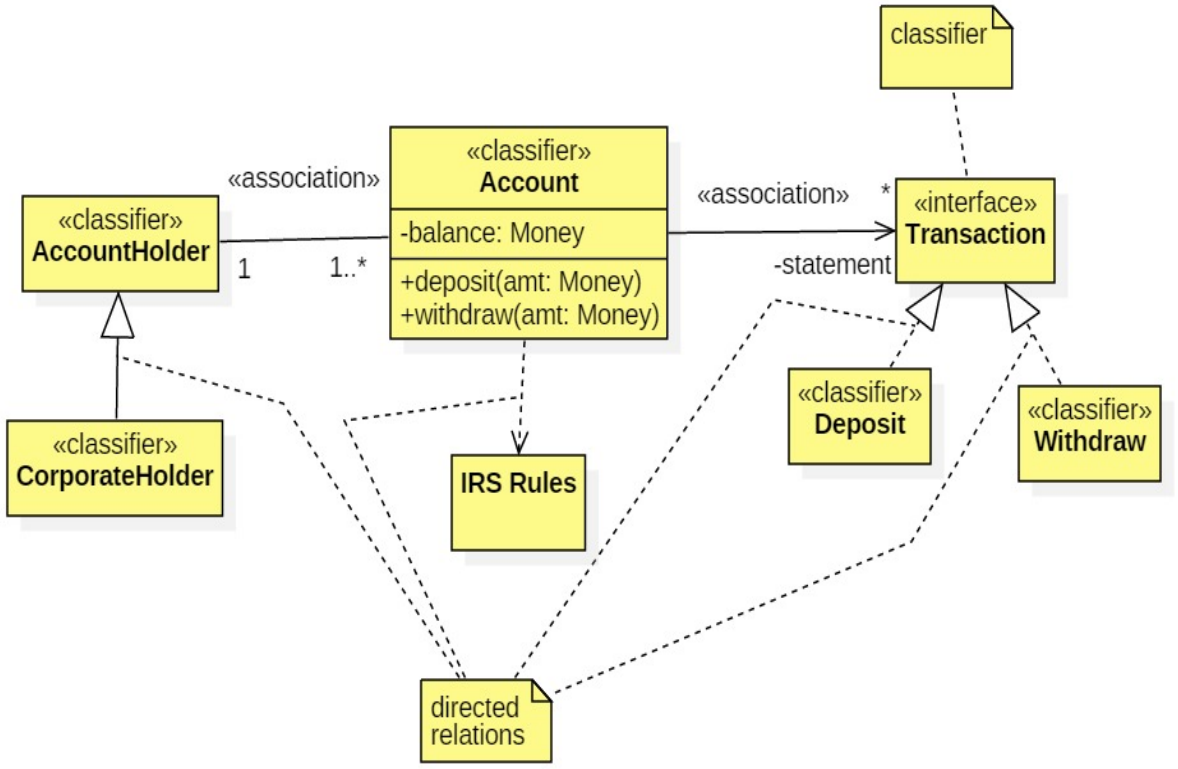
An (binary) association represents a semantic relationship between (two) classes. A semantic relationship is a relationship is a domain-significant relationship. For example, there could be many relationships between two people—teacher/student, parent/child, employee/dependent, etc. But in a genealogy model, for example, we are probably only interested in the parent/child relationship.

An instance of a parent/child association would be a link connecting an instance of person representing a parent to an instance of person representing one of the parent's children. For example:



Here are a few more examples:



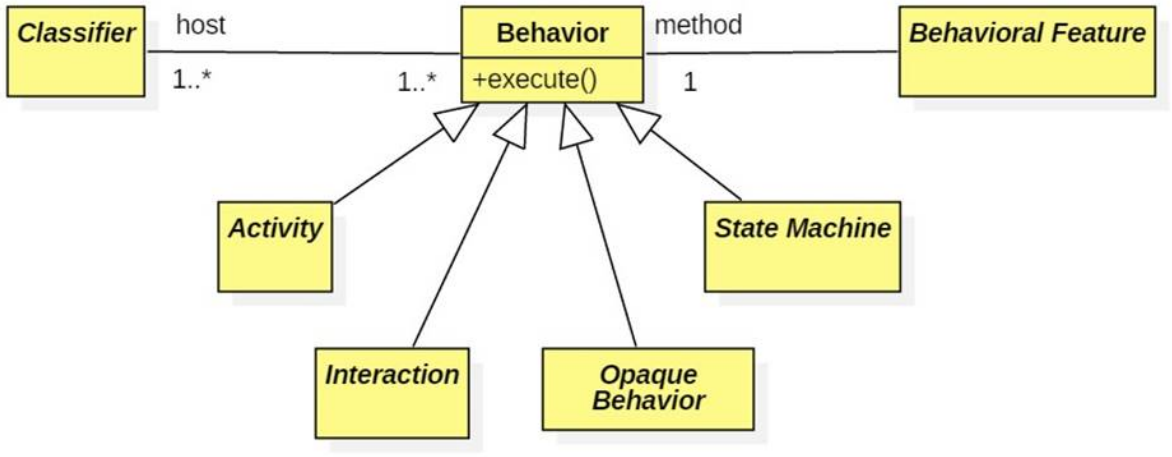


A classifier can have many features. Features can be structural or behavioral. Attributes are examples of structural features for a class (e.g., an account's balance). Operations are examples of behavioral features for a class (e.g., deposit and withdraw for accounts).

Behaviors

A behaviored classifier is one that can be associated with one or more behaviors. An instance of the classifier (called the host) executes a behavior. A behavioral feature such as an operation is associated with a behavior representing the operations implementation. This behavior is executed when the operation is invoked.

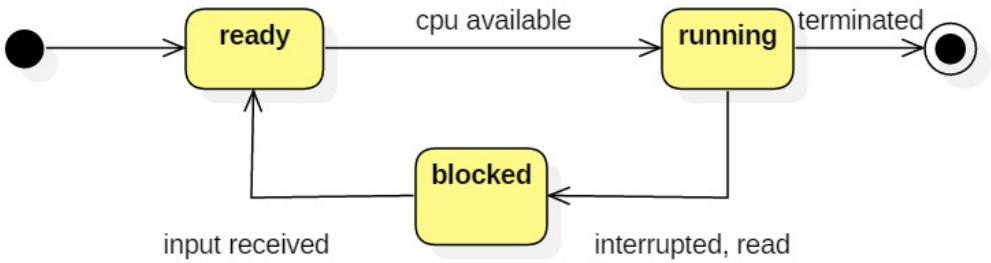
There are four basic types of behaviors:



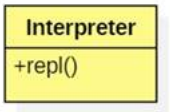
An opaque behavior is one with implementation-specific semantics, like a block of C++ code.

State Machine Behaviors

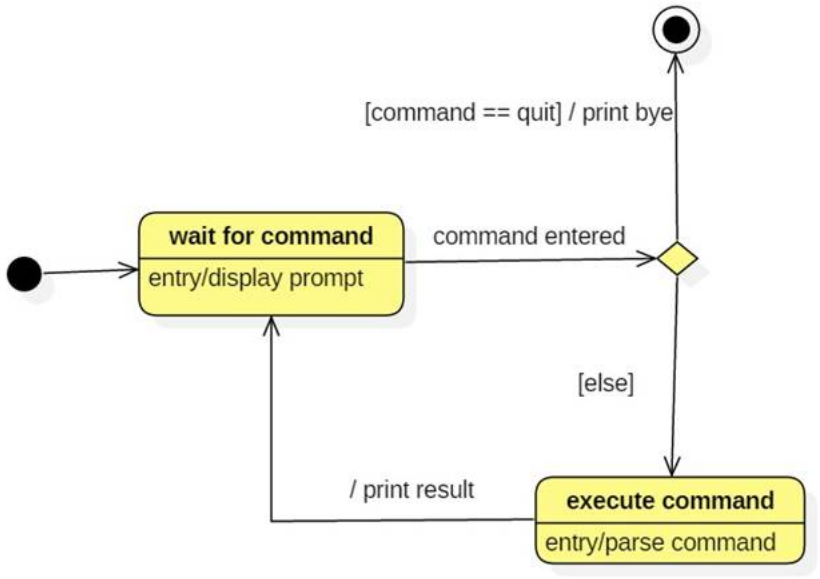
Some objects have well defined lifecycles which can be described using a state machine, For example, the lifecycle of a process:



Usually the behavior of an operation is opaque, given by some language-specific block of code. In some cases the behavior could be specified by a state machine. As an example, assume Interpreter is a class with a read-execute-print loop method that perpetually prompts a user for a command, executes the command, then prints the result.

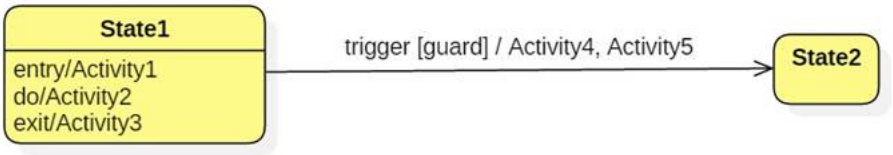


The repl() operation could be associated with a state machine behavior:



In a state machine arrows represent transitions between states (and are a form of directed relationship). States and transitions can also have behaviors. For example, when a classifier instance is in a state it may execute behaviors upon entry, while in the state, and upon exiting the state. These behaviors can be activities, state machines, interactions, or opaque.

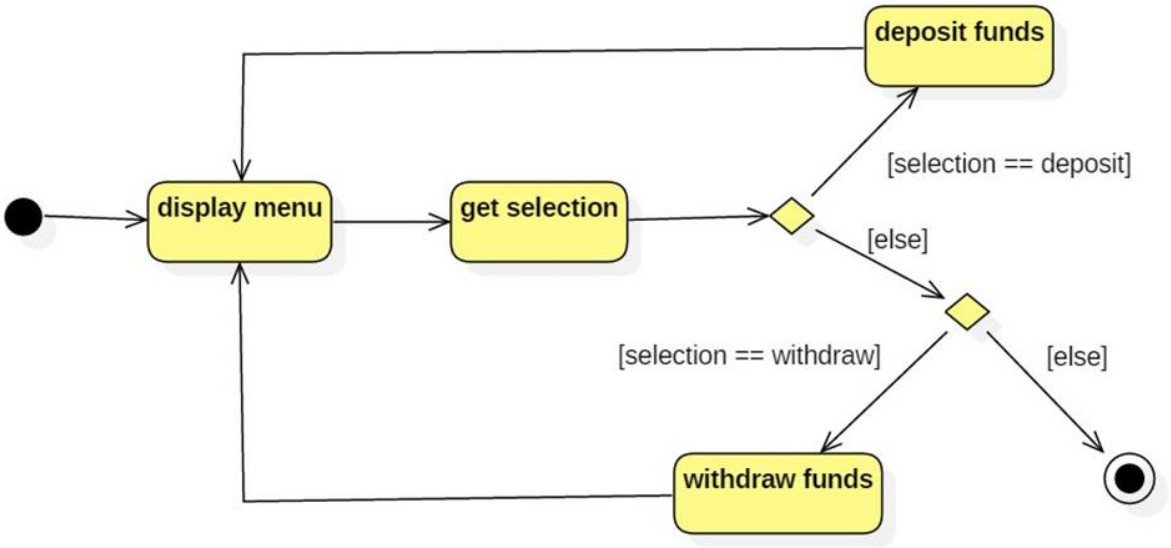
A transition is triggered by an event (signal, message, change, or time). A transition may have a Boolean guard that must be satisfied before the transition occurs. In addition, the transition may have execute several behaviors (called effects).



Activities

Activities can be composite or simple. A simple activity is called an action. A composite activity consists of activity or action nodes connected by control flow arrows (another type of directed relationship).

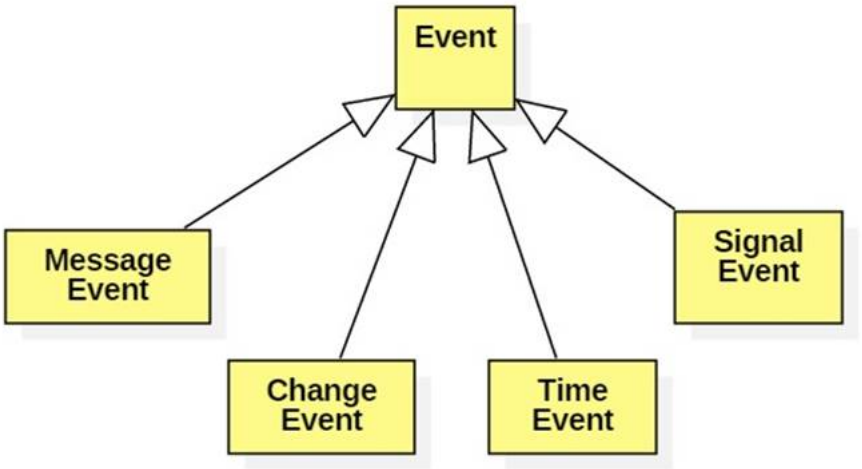
For example, the following ATM machine activity consists of four actions (not counting the decision nodes):



Kinds of actions include: create, destroy, read, write, insert, delete, send signal, receive signal, trigger event, accept event, timing event, and opaque. In the example above display menu might be a write action while get selection might be a read action.

Events

The execution of a behavior is triggered by an event.



A time even occurs at a specified time. A timeout is an example.

A message event means the classifier instance has received a message such as a method call.

A change event means some object has changed state.

A signal even is the reception of an asynchronous broadcast signal.

Outline Development Process

Unified Process is based on the enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation. The system is developed incrementally over time, iteration by iteration, and thus this approach is also known as iterative and incremental software development. The iterations are spread over four phases where each phase consists of one or more iterations [4]:

Inception—the first and the shortest phase in the project. It is used to prepare basis for the project, including preparation of business case, establishing project scope and setting boundaries, outlining key requirements, and possible architecture solution together with design tradeoffs, identifying risks, and development of initial project plan—schedule with main milestones and cost estimates. If the inception phase lasts for too long, it is like an indicator stating that the project vision and goals are not clear to the stakeholders. With no clear goals and vision the project most likely is doomed to fail. At this scenario it is better to take a pause at the very beginning of the project to refine the vision and goals. Otherwise it could lead to unnecessary make-overs and schedule delays in further phases.

Elaboration—during this phase the project team is expected to capture a majority of system's requirements (e.g., in the form of use cases), to perform identified risk analysis and make a plan of risk management to reduce or eliminate their impact on final schedule and product, to establish design and architecture (e.g., using basic class diagrams, package diagrams, and deployment diagrams), to create a plan (schedule, cost estimates, and achievable milestones) for the next (construction) phase.

Construction—the longest and largest phase within Unified Process. During this phase, the design of the system is finalized and refined and the system is built using the basis created during elaboration phase. The construction phase is divided into multiple iterations, for each iteration to result in an executable release of the system. The final iteration of

construction phase releases fully completed system which is to be deployed during transition phase, and

Transition—the final project phase which delivers the new system to its end-users. Transition phase includes also data migration from legacy systems and user trainings.

Each phase and its iteration consists of a set of predefined activities. The Unified Process describes work activities as disciplines—a discipline is a set of activities and related artifacts in one subject area (e.g., the activities within requirements analysis). The disciplines described by Unified Process are as follows [107]:

Business modeling—domain object modeling and dynamic modeling of the business processes,

Requirements—requirements analysis of system under consideration. Includes activities like writing use cases and identifying nonfunctional requirements,

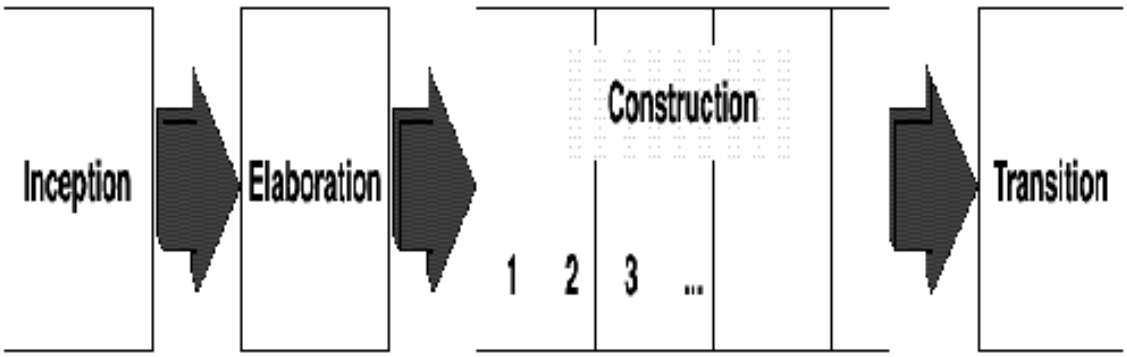
Analysis and design—covers aspects of design, including the overall architecture,

Implementation—programming and building the system (except the deployment),

Test—involves testing activities such as test planning, development of test scenarios, alpha and beta testing, regression testing, acceptance testing, and

-

This process is an iterative and incremental development process, in that the software is not released in one big bang at the end of the project but is, instead, developed and released in pieces. The **construction** phase consists of many **iterations,** in which each iteration builds production-quality software, tested and integrated, that satisfies a



subset of the requirements of the project. The delivery may be external, to early users, or purely internal. Each iteration contains all the usual life-cycle phases of analysis, design, implementation, and testing.

In principle, you can start at the beginning: Pick some functionality and build it, pick some other functionality, and so forth. However, it is worthwhile to spend some time planning.

The first two phases are inception and elaboration. During **inception,** you establish the business rationale for the project and decide on the scope of the project. This is where you get the commitment from the project sponsor to go further. In **elaboration,** you collect more detailed requirements, do high-level analysis and design to establish a baseline architecture, and create the plan for construction.

Even with this kind of iterative process, some work has to be left to the end, in the **transition** phase. This work can include beta testing, performance tuning, and user training.

Projects vary in how much **ceremony** they have. High-ceremony projects have a lot of formal paper deliverables, formal meetings, formal sign-offs. Low-ceremony projects might have an inception phase that consists of an hour's chat with the project's sponsor and a plan that sits on a spreadsheet. Naturally, the bigger the project, the more ceremony you need. The fundamentals of the phases still occur, but in very different ways.

I try to keep the ceremony to a minimum, and my discussion reflects that. There are plenty of high-ceremony processes to choose from elsewhere.

I've shown iterations in the construction phase, but not in the other phases. In fact, you can have iterations in all phases, and it is often a good idea to do so in a large phase. Construction is the key phase in which to iterate, however.

That's the high-level view. Now we will delve into the details so that we have enough information to see where the techniques discussed later in the book fit into the larger scheme of things. In doing this, I will talk a bit about these techniques and when to use them. You may find it a little confusing if you are unfamiliar with the techniques. If that's the case, skip those bits and come back to them later.

Inception

Inception can take many forms. For some projects, it's a chat at the coffee machine: "Have a look at putting our catalog of services on the Web." For bigger projects, it might be a full-fledged feasibility study that takes months.

During the inception phase, you work out the business case for the projectroughly how much it will cost and how much it will bring in. You will also need to get a sense of the project's scope. You may need to do some initial analysis to get a sense of the size of the project.

I don't tend to make a big deal of inception. Inception should be a few days' work to consider whether it is worth doing a few months' worth of deeper investigation during elaboration

Deployment—the deployment activities of developed system.

The UML is a modeling language, not a method. The UML has no notion of process, which is an important part of a method.

The three amigos have developed a merged process called the *Rational Unified Process*. (It used to be called Objectory.) This process is described in the amigos' process book (Jacobson, Booch, and Rumbaugh 1999).

The UML is independent of process. You should pick something that is appropriate for your kind of project. Whatever process you use, you can use the UML to record the resulting analysis and design decisions.

Inception

Inception can take many forms. For bigger projects, it might be a full-fledged feasibility study that takes months.

During the inception phase, you work out the business case for the project roughly how much it will cost and how much it will bring in. You will also need to get a sense of the project's scope. You may need to do some initial analysis to get a sense of the size of the project.

Inception should be a few days' work to consider whether it is worth doing a few months' worth of deeper investigation during elaboration (see the next section). At this point, the project's sponsor agrees to no more than a serious look at the project.

Elaboration

In this stage, typically, we have only a vague idea of the requirements. For instance, we might be able to say:

We are going to build the next-generation customer support system for the Watts Galore Utility Company. We intend to use object-oriented technology to build a more flexible system that is more customer-orientedspecifically, one that will support consolidated customer bills.

Of course, your requirements document will likely be more expansive than that, but it may not actually say very much more.

At this point, we want to get a better understanding of the problem.

- What is it you are actually going to build?
- How are you going to build it?

In deciding what issues to look into during this phase, we need to be driven, first and foremost, by the risks in your project. What things could derail you? The bigger the risk, the more attention you have to pay to it.

The risks can usefully be classified into four categories:

1. **Requirements risks.**

   What are the requirements of the system? The big danger is that you will build the wrong system, one that does not do what the customer needs.

2. **Technological risks.**

   What are the technological risks you have to face? Are you selecting technology that will actually do the job for you? Will the various pieces fit together?

3. **Skills risks.**

Can you get the staff and expertise you need?

4. **Political risks.**

   Are there political forces that can get in the way and seriously affect your project?

There may be more in your case, but risks that fall into these four categories are nearly always present.

Dealing with Requirements Risks

Requirements are important and are where UML techniques can most obviously be brought to bear. The starting point is use cases. Use cases drive the whole development process.

A use case is a typical interaction that a user has with the system in order to achieve a goal. Imagine the word processor that I am currently using. One use case would be "do a spell check"; another would be "create an index for a document."

The key element for a use case is that each one indicates a function that the user can understand and that has value for that user. A developer can respond with specifics.

Use cases provide the basis of communication between customers and developers in planning the project.

One of the most important things to do in the elaboration phase is to discover all the potential use cases for the system you are building. In practice, of course, you aren't going to get all of them. You want to get most, however, particularly the most important and riskiest ones. It's for this reason that, during the elaboration phase, you should schedule interviews with users for the purpose of gathering use cases.

Use cases do not need to be detailed. I usually find that a paragraph or three of descriptive text is sufficient. This text should be specific enough for the users to understand the basic idea and for the developers to have a broad sense of what lurks inside.

Use cases are not the whole picture, however. Another important task is to come up with the skeleton of a conceptual model of the domain. Within the heads of one or more users lies a picture of how the business operates. For instance:

Our customers may have several sites, and we provide several services to these sites. At the moment, a customer gets a bill for all services at a given site. We want that customer to be billed for all services at all sites. We call this consolidated billing.

The main technique to use for domain models is the class diagram, drawn from a conceptual perspective. You can use these diagrams to lay out the concepts that the business experts use as they think about the business and to lay out the ways those experts link concepts together. In many ways, class diagrams are about defining a rigorous vocabulary to talk about the domain.

If the domain also has a strong workflow element, we have to describe this with activity diagrams. The key aspect of activity diagrams is that they encourage finding parallel processes, which is important in eliminating unnecessary sequences in business processes.

Domain modeling can be a great adjunct to use cases. When gathering use cases, we have to bring in a domain expert and explore how that person thinks about the business, with the help of conceptual class diagrams and activity diagrams.

In this situation, we have to use minimal notation, don't worry about rigor, and we make lots of informational notes on the diagram. don't try to capture every detail. Instead, focus on important issues and areas that imply risk. We may draw lots of unconnected diagrams without worrying about consistency and interrelationships among diagrams.

One of the most important elements in dealing with requirements risk is getting access to domain expertise. Lack of access to people who really know the domain is one of the commonest ways for projects to fail. It is worth investing considerable time and money to bring people who really know the domain into your team the quality of the software will be directly proportional to their expertise. They need not be full time, but they need to be open-minded, have deep hands-on understanding, and be readily available for questions.

Dealing with Technological Risks

The most important thing to do in addressing technological risks is to build prototypes that try out the pieces of technology you are thinking of using.

For example, say you are using C++ and a relational database. You should build a simple application using C++ and the database together. Try out several tools and see which ones work best. Spend some time getting comfortable with the tools you are going to use.

Don't forget that the biggest technological risks are inherent in how the components of a design fit together rather than being present in any of the components themselves. You may know C++ well, and you may know relational databases well, but putting them together can be surprisingly hard. This is why it is very important to get all the components you intend to use and fit them together at this early stage of the process.

You should also address any architectural design decisions during this stage. These usually take the form of ideas of what the major components are and how they will be built. This is particularly important if you are contemplating a distributed system.

As part of this exercise, focus on any areas that look as though they will be difficult to change later. Try to do your design in a way that will allow you to change elements of the design relatively easily. Ask yourself these questions.

- What will happen if a piece of technology doesn't work?
- What if we can't connect two pieces of the puzzle?
- What is the likelihood of something going wrong? How would we cope if that happens?

As with the domain model, you should look at the use cases as they appear in order to assess whether they contain anything that could cripple your design. If you fear they may contain a "purple worm," investigate further.

During this process, you will typically use a number of UML techniques to sketch out your ideas and document the things you try. Don't try to be comprehensive at this point; brief sketches are all you need and, therefore, all you should use.

- Class diagrams and interaction diagrams are useful in showing how components communicate.
- Package diagrams can show a high-level picture of the components at this stage.
- Deployment diagrams can provide an overview of how pieces are distributed.

Dealing with Skills Risks

Training is a way to avoid making mistakes, because instructors have already made those mistakes. Making mistakes takes time, and time costs money. So you pay the same either way, but not having the training causes the project to take longer.

A short training course can be useful, but it's only a beginning.If you do go for a short training course, pay a lot of attention to the instructor. It is worth paying a lot extra for someone who is knowledgeable and entertaining, because you will learn a lot more in the process. Also, get your training in small chunks, just at the time you need it. If you don't apply what you have learned in a training course straight away, you will forget it.

The best way to acquire OO skills is through **mentoring**, in which you have an experienced developer work with your project for an extended period of time. The mentor shows you how to do things, watches what you do, and passes on tips and short bits of training.

A mentor will work with the specifics of your project and knows which bits of expertise to apply at the right time. In the early stages, a mentor is one of the team, helping you come up with a solution. As time goes on, you become more capable, and the mentor does more reviewing than doing. My goal as a mentor is to render myself unnecessary.

You can find mentors for specific areas or for the overall project. Mentors can be full time or part time. Many mentors like to work a week out of each month on each project; others find that too little. Look for a mentor with knowledge and the ability to transfer that knowledge. Your mentor may be the most important factor in your project's success; it is worth paying for quality.

If you can't get a mentor, consider a project review every couple of months or so. Under this setup, an experienced mentor comes in for a few days to review various aspects of the design. During this time, the reviewer can highlight any areas of concern, suggest additional ideas, and outline any useful techniques that the team may be unaware of. Although this does not give you the full benefits of a good mentor, it can be valuable in spotting key things that you can do better.

You can also supplement your skills by reading. Try to read a solid technical book at least once every other month. Even better, read it as part of a book group. Find a couple of other people who want to read the same book. Agree to read a few chapters a week, and spend an hour or two discussing those chapters with the others. By doing this, you can gain a better understanding of the book than by reading it on your own. If you are a manager, encourage this. Get a room for the group; give your staff the money to buy technical books; allocate time for a book group.

The patterns community has found book groups to be particularly valuable. Several patterns reading groups have appeared.

As you work through elaboration, keep an eye out for any areas in which you have no skills or experience. Plan to acquire the experience at the point at which you need it.

When Is Elaboration Finished?

elaboration takes about a fifth of the total length of the project. Two events are key indicators that elaboration is complete.

- The developers can feel comfortable providing estimates, to the nearest person-week of effort, of how long it will take to build each use case.

- All the significant risks have been identified, and the major ones are understood to the extent that you know how you intend to deal with them.

Planning the Construction Phase

There are many ways to plan an iterative project. It's important that you develop a plan in order to be aware of progress and to signal progress through the team.

The essence of building a plan involves setting up a series of iterations for construction and defining the functionality to deliver in each iteration. Some people like to use small use cases and complete a use case within an iteration; others like to use larger use cases, doing some scenarios in one iteration and others later. The basic process is the same. describe it with the smaller use cases.

During planning, consider two groups of people: customers and developers.

Customers are the people who are going to use the system for an inhouse development. For a shrink-wrap system, marketing people usually represent the customer. The key thing is that the customers are the people who can assess the business value of a use case being implemented.

The developers are the people who are going to build the system. They must understand the costs and effort involved in building a use case. So, they must be familiar with the development environment. Management usually cannot play this role, because you need recent technical experience to do this.

The first step is to categorize the use cases. I do this two ways.

First, the customer divides the use cases, according to their business value, into three piles: high, medium, and low. (Note that it's considered bad form to put everything in the "high" pile.) Then the customer records the contents of each category.

The developers then divide the use cases according to the development risk. For instance, "high risk" would be used for something that is very difficult to do, could have a big impact on the design of the system, or is just not well understood.

After this is done, the developers should estimate the length of time each use case will require, to the nearest person-week. In performing this estimate, assume that you need to do analysis, design, coding, unit testing, integration, and documentation. Assume also that you have a fully committed developer with no distractions (we'll add a fudge factor later).

Once your estimates are in place, you can assess whether you are ready to make the plan. Look at the use cases with high risk. If a lot of the project's time is tied up in these use cases, you need to do more elaboration.

The next step is to determine your iteration length. You want a fixed iteration length for the whole project so that you get a regular rhythm to the iteration delivery. An iteration should be long enough for you to do a handful of use cases. For Smalltalk, it can be as low as two to three weeks, for instance; for C++, it can be as high as six to eight weeks.

Now you can consider how much effort you have for each iteration.

Note that you will have made these estimates assuming a developer with no distractions. Obviously, this is never the case, so I allow for that with a load factor that is the difference between ideal time and the reality. You should measure this load factor by comparing estimates to actuals.

Now you can work out how fast you can go, which I term the project velocity. This is how much development you can do in an iteration. You calculate this by taking your number of developers, multiplying it by the iteration length, and then dividing the result by the load factor. For instance, given 8 developers, a 3-week iteration length, and a load factor of 2, you would have 12 ideal developer-weeks (8 * 3 * 1/2) of effort per iteration.

Add up your time for all use cases, divide by the effort per iteration, and add 1 for luck. The result is your first estimate of how many iterations you will need for your project.

The next step is to assign the use cases to iterations.

Use cases that carry high priority and/or development risk should be dealt with early. Do *not* put off risk until the end! You may need to split big use cases, and you will probably revise use case estimates in light of the order in which you are doing things. You can have less work to do than the effort in the iteration, but you should never schedule more than your effort allows.

For transition, allocate from 10 percent to 35 percent of the construction time for tuning and packaging for the delivery. (Use a higher figure if you are inexperienced with tuning and packaging in your current environment.)

Then add a contingency factor: 10 percent to 20 percent of the construction time, depending on how risky things look. Add this factor to the end of the transition phase. You should plan to deliver without using contingency timethat is, on your internal target datebut commit to deliver at the end of contingent time.

After following all of these guidelines, you should have a **release plan** that shows the use cases that will be done during each iteration. This plan symbolizes commitment among developers and users. This plan is not cast in stoneindeed, everyone should expect the plan to change as the project proceeds. Since it is a commitment between developers and users, however, changes must be made jointly.

As you can see from this discussion, use cases serve as the foundation for planning the project, which is why the UML puts a lot of emphasis on them.

Construction

Construction builds the system in a series of iterations. Each iteration is a mini-project. You do analysis, design, coding, testing, and integration for the use cases assigned to each iteration. You finish the iteration with a demo to the user and perform system tests to confirm that the use cases have been built correctly.

The purpose of this process is to reduce risk. Risk often appears because difficult issues are left to the end of the project. In many Projects,testing and integration are left to the end. Testing and integration are big tasks, and they always take longer than people think. Left to the end, they are hard and demoralizing. That's why we should always encourage the clients to develop self-testing software.

The iterations within construction are both incremental and iterative

- The iterations are *incremental* in function. Each iteration builds on the use cases developed in the previous iterations.
- The iterations are *iterative* in terms of the code base. Each iteration will involve rewriting some existing code to make it more flexible.

Refactoring

Refactoring is a highly useful technique in iterating the code. It's a good idea to keep an eye on the amount of code thrown away in each iteration. Be suspicious if less than 10 percent of the previous code is discarded each time.

Integration should be a continuous process. For starters, full integration is part of the end of each iteration. However, integration can and should occur more frequently than that. A good practice is to do a full build and integration every day. By doing that every day, things never get so far out of sync that it becomes a problem to integrate them later.

Refactoring is a term used to describe techniques that reduce the short-term pain of redesigning. When you refactor, you do not change the functionality of your program; rather, you change its internal structure in order to make it easier to understand and work with.

Refactoring changes are usually small steps: renaming a method, moving a field from one class to another, consolidating two similar methods into a superclass. Each step is tiny, yet a couple of hours' worth of performing these small steps can do a world of good to a program.

Refactoring is made easier by the following principles.

- Do not refactor a program and add functionality to it at the same time. Impose a clear separation between the two when you work. You might swap between them in short stepsfor instance, half an hour refactoring, an hour adding a new function, and half an hour refactoring the code you just added.
- Make sure you have good tests in place before you begin refactoring.
- Take short, deliberate steps. Move a field from one class to another. Fuse two similar methods into a superclass. Test after each step. This may sound slow, but it avoids debugging and thus speeds you up.

You should refactor when you are adding a new function or fixing a bug. Don't set aside specific time for refactoring; instead, do a little every day.

Using the UML in Construction

All UML techniques are useful during this stage.

A conceptual class diagram can be useful to rough out some concepts for the use case and see how these concepts fit with the software that has already been built.

The advantage of these techniques at this stage is that they can be used in conjunction with the domain expert. As Brad Kain says: Analysis occurs only when the domain expert is in the room (otherwise it is pseudo-analysis).

To make the move to design, walk through how the classes will collaborate to implement the functionality required by each use case. CRC cards and interaction diagrams are useful in exploring these interactions. These will expose responsibilities and operations that you can record on the class diagram.

Treat these designs as an initial sketch and as a tool with which to discuss design approaches with your colleagues. Once you are comfortable, it is time to move to code.

Inevitably, the unforgiving code will expose weaknesses in the design. Don't be afraid to change the design in response to this learning. If the change is serious, use the notations to discuss ideas with your colleagues.

Once you have built the software, you can use the UML to help document what you have done. For this, I find UML diagrams useful for getting an overall understanding of a system. In doing this, however, I should stress that I do not believe in producing detailed diagrams of the whole system. To quote Ward Cunningham (1996):

Carefully selected and well-written memos can easily substitute for traditional comprehensive design documentation. The latter rarely shines except in isolated spots. Elevate those spots... and forget about the rest.

detailed documentation should be generated from the code (like, for instance, JavaDoc). You should write additional documentation to highlight important concepts. Think of these as comprising a first step for the reader before he or she goes into the code-based details. We have to structure these as prose documents, short enough to read over a cup of coffee, using UML diagrams to help illustrate the discussion.

We may use a package diagram as a logical road map of the system. This diagram helps tounderstand the logical pieces of the system and see the dependencies (and keep them under control). A deployment diagram, which shows the high-level physical picture, may also prove useful at this stage.

Within each package, we have to see a specification-perspective class diagram. don't show every operation on every class. show only the associations and key attributes and operations that help me understand what is in there. This class diagram acts as a graphical table of contents.

If a class has complex lifecycle behavior, draw a state diagram to describe it. I do this only if the behavior is sufficiently complex, which doesn't happen often. More common are complicated interactions among classes, for which I draw interaction diagrams.

We may include some important code, written in a literate program style. If a particularly complex algorithm is involved, we may consider using an activity diagram, but only if it gives me more understanding than the code alone.

If we find concepts that are coming up repeatedly, we may use patterns to capture the basic ideas.

Transition

The point of iterative development is to do the whole development process regularly so that the development team gets used to delivering finished code. But some things should not be done early. A prime example is optimization.

Optimization reduces the clarity and extensibility of the system in order to improve performance. That is a trade-off you need to make after all, a system does have to be fast enough to meet users' requirements. But optimizing too early makes development tougher, so this is one thing that does need to be left to the end.

During transition, there is no development to add functionality, unless it is small and absolutely essential. There is development to fix bugs. A good example of a transition phase is that time between the beta release and the final release of a product.

When to Use Iterative Development

You should use iterative development only on projects that you want to succeed.

Perhaps that's a bit glib, but as I get older, I get more aggressive about using iterative development. Done well, it is an essential technique, which you can use to expose risk early and to obtain better control over development. It is not the same as having no management (although, to be fair, I should point out that some have used it that way). It does need to be well planned. But it is a solid approach, and every OO development book encourages using itfor good reason.

Patterns

The UML tells you how to express an object-oriented design. **Patterns** look, instead, at the results of the process: example models.

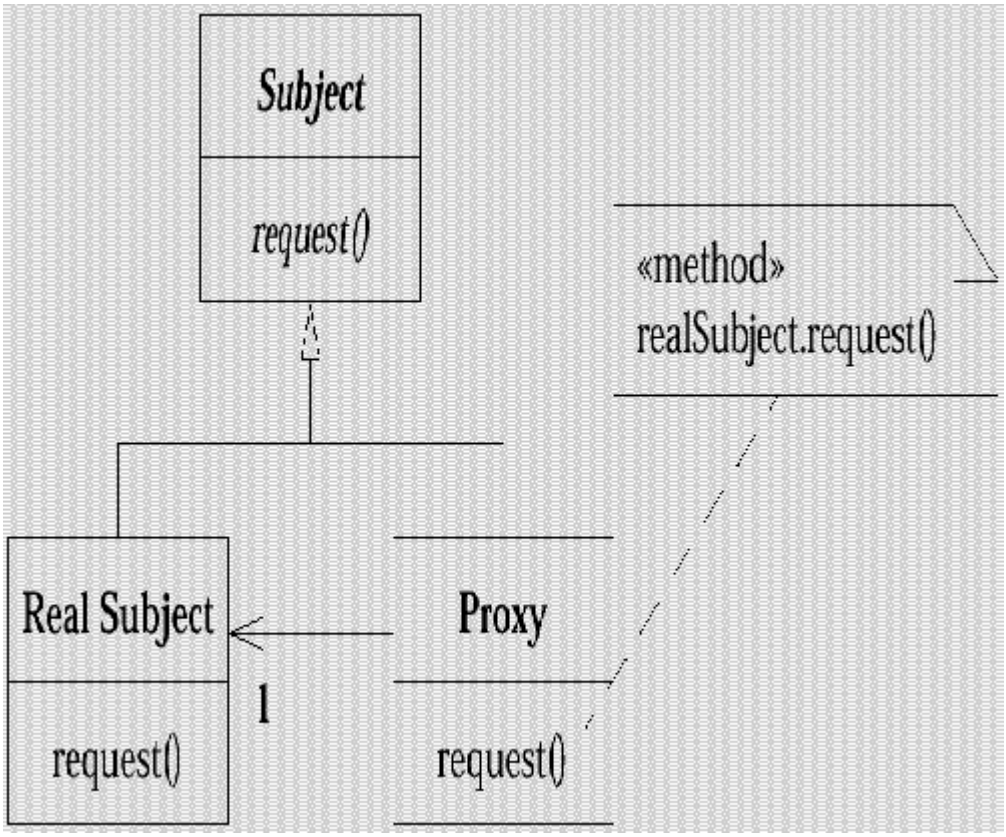
Many people have commented that projects have problems because the people involved were not aware of designs that are well known to those with more experience. Patterns describe common ways of doing things. They are collected by people who spot repeating themes in designs. These people take each theme and describe it so that other people can read the pattern and see how to apply it.

Let's look at an example. Say you have some objects running in a process on your desktop, and they need to communicate with other objects running in another process. Perhaps this process is also on your desktop; perhaps it resides elsewhere. You don't want the objects in your system to have to worry about finding other objects on the network or executing remote procedure calls.

What you can do is create a *proxy* object within your local process for the remote object. The proxy has the same interface as the remote object. Your local objects talk to the proxy using the usual in-process message sends. The proxy then is responsible for passing any messages on to the real object, wherever it might reside.

Figure 2-2 is a class diagram (see Chapter 4) that illustrates the structure of the *Proxy* pattern.

Figure 2-2. Structure of Proxy Design Pattern



Proxies are a common technique used in networks and elsewhere. People have a lot of experience using proxies in terms of knowing how they can be used, what advantages they can bring, their limitations, and how to implement them. Methods books like this one don't discuss this knowledge; all they discuss is how you can diagram a proxy. Although useful, it is not as useful as discussing the experience involving proxies.

In the early 1990s, some people began to capture this experience. They formed a community interested in writing patterns. These people sponsor conferences and have produced several books.

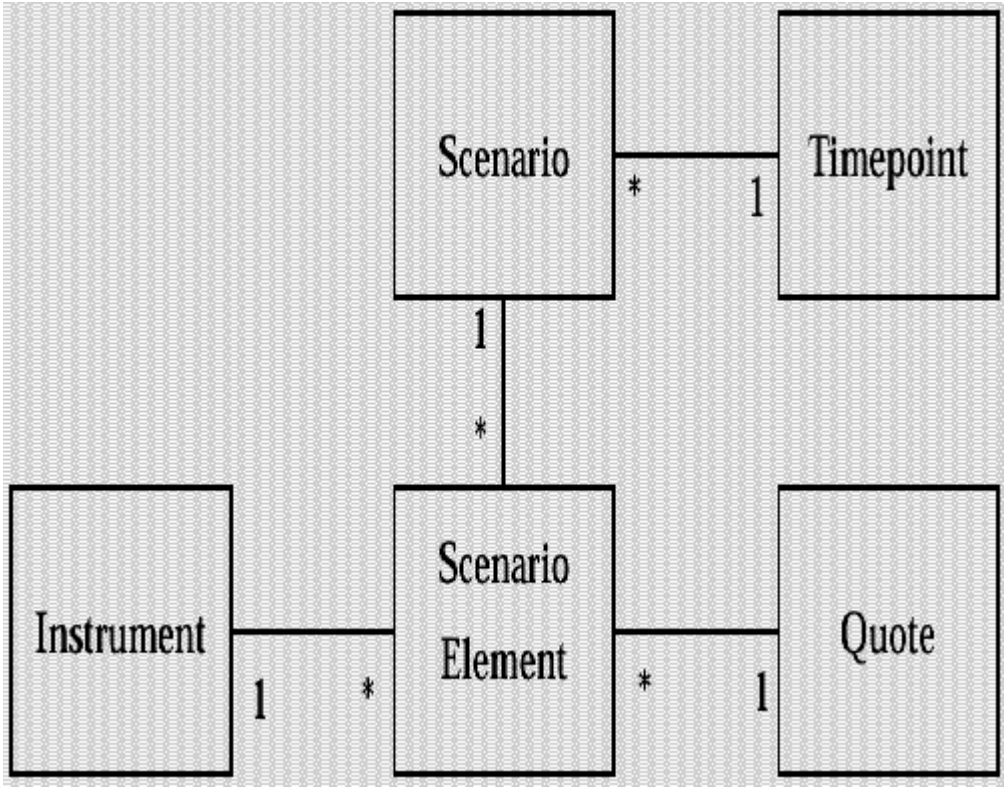
The most famous patterns book to emerge from this group is the Gang of Four book (Gamma, Helm, Johnson, and Vlissides 1995), which discusses 23 design patterns in detail.

If you want to know about proxies, this is the source. The Gang of Four book spends 10 pages on the subject, giving details about how the objects work together, the benefits and limitations of the pattern, common variations and usages, and implementation tips for Smalltalk and C++.

Proxy is a design pattern because it describes a design technique. Patterns can also exist in other areas. Say you are designing a system for managing risk in financial markets. You need to understand how the value of a portfolio of stocks changes over time. You could do this by keeping a price for each stock and timestamping the price. However, you also want to be able to consider the risk in hypothetical situations (for instance, "What will happen if the price of oil collapses?").

To do this, you can create a *Scenario* that contains a whole set of prices for stocks. Then you can have separate *Scenarios* for the prices last week, your best guess for next week, your guess for next week if oil prices collapse, and so forth. This *Scenario* pattern (see Figure 2-3) is an analysis pattern because it describes a piece of domain modeling.

Figure 2-3. Scenario Analysis Pattern



Analysis patterns are valuable because they give you a better start when you work with a new domain. I started collecting analysis patterns because I was frustrated by new domains. I knew I wasn't the first person to model them, yet each time I had to start with a blank sheet of paper.

The interesting thing about analysis patterns is that they crop up in unusual places. When I started working on a project to do corporate financial analysis, I was enormously helped by a set of patterns I had previously discovered in health care.

See Fowler (1997) to learn more about *Scenario* and other analysis patterns.

A pattern is much more than a model. A pattern must also include the reason why it is the way it is. It is often said that a pattern is a solution to a problem. The pattern must make the problem clear, explain why it solves the problem, and also explain in what circumstances it works and does not work.

Patterns are important because they are the next stage beyond understanding the basics of a language or a modeling technique. Patterns give you a series of solutions and also show you what makes a good model and how you go about constructing a model. They teach by example.

UNIT-IV

OBJECT ORIENTED DESIGN

OO Design Process and Design Axioms

OBJECTIVES:

At the end of this chapter, students should be able to

- Define and understand the object-oriented design process
- Explain the object-oriented design rules

COURSE OUTCOME

- Objects discovered serve as the framework for design.

- Identified objects, attributes, methods, associations must be designed for implementation as a data type expressed in the implementation language

INTRODUCTION

- Main focus of the analysis phase of SW development ➔ "what needs to be done"
- During the design phase, we elevate the model into logical entities, some of which might relate more to the computer domain (such as user interface, or the access layer) than the real world or the physical domain (such as people or employees). Start thinking how to actually implement the problem in a program.
- The goal ➔ to design the classes that we need to implement the system.
- Design is about producing a solution that meets the requirements that have been specified during analysis.

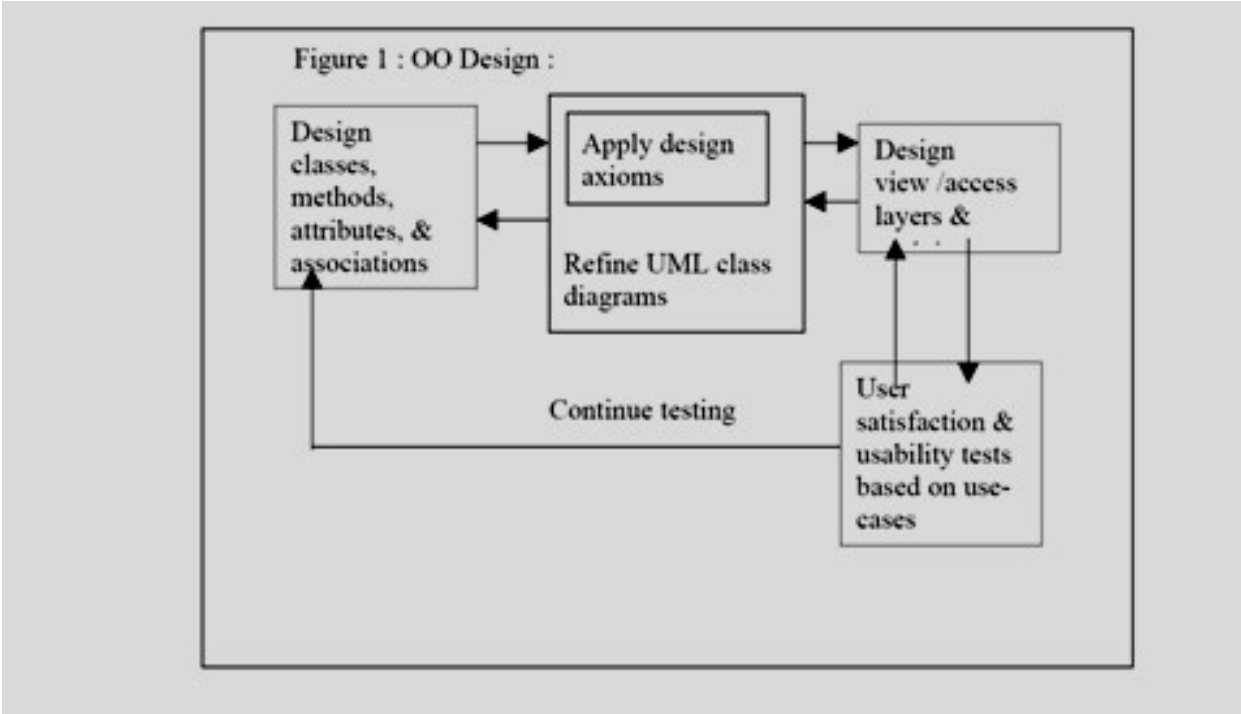
Object-Oriented Design Process and Design Axioms

- Analysis Phase
  - Class's attributes, methods and associations are identified
  - Physical entities, players and their cooperation are identified
  - Objects can be individuals, organizations or machines
- Design Phase
  - Using Implementation language appropriate data types are assigned
  - Elevate the model into logical entities (user interfaces)
  - Focus is on the view and access classes (How to maintain information or best way to interact with a user)

Importance of Good Design

- Time spent on design decides the success of the software developed.
- Good design simplifies implementation and maintenance of a project.
- To formalize (celebrate, honor) design process, axiomatic (clear) approach is to be followed

OO design process



Activities of OOD Process

1. Apply design axioms to design classes, their attributes, methods, associations, structure and protocols.

1.1 Refine and complete the static UML class diagram by adding details to the UML class diagram. This steps consists of the following activities:

Refine attributes

Design methods and protocols by utilizing a UML activity diagram to represent the methods algorithm.

Refine association between classes (if required)

Refine class hierarchy and design with inheritance (if required).

1.2 Iterate and refine again.

(2) Design the access layer

- Create mirror classes: For every business class identified and created, create one access layer
- . Eg , if there are 3 business classes (class1, class2 and class3), create 3 access layer classes (class1DB, class2DB and class3DB)
- Identify access layer class relationships

Simplify classes and their relationships – main goal is to eliminate redundant classes and structures

- Redundant classes: Do not keep 2 classes that perform similar translate request and translate results activities. Select one and eliminate the other.
- Method classes: Revisit the classes that consist of only one or two methods to see if they can be eliminated or combined with existing classes.

- **Iterate and refine again.**

(3) Design the view layer classes
  - Design the macro level user interface, identifying view layer objects
  - Design the micro level user interface, which includes these activities:

(a) Design the view layer objects by applying the design axioms and corollaries.

(b) Build a prototype of the view layer interface
  - Test usability and user satisfaction
  - Iterate and refine

(4) Iterate and refine the whole design. Reapply the design axioms and if needed, repeat the preceding steps.

- An axiom is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception.
- They cannot be proven or derived but they can be invalidated by counterexamples or exceptions.
- A theorem is a proposition that may not be self-evident (clear) but can be proved from accepted axioms.
- A corollary is a proposition that follows from an axiom or another proposition that has been proven.

Types of Axioms
- AXIOM-1 (Independence axiom): deals with relationships between system components such as classes, requirements, software components.
- AXIOM-2 (Information axiom): deals with the complexity of design

Axioms of OOD
- The axiom 1 of object-oriented design deals with relationships between system components (such as classes, requirements and software components) and axiom 2 deals with the complexity of design.
- Axiom 1. The independence axiom. Maintain the independence of components. According to axiom 1, each component must satisfy its requirements without affecting other requirements. Eg. Let us design a refrigerator door which can provide access to food and the energy lost should be minimized when the door is opened and closed. Opening the door should be independent of losing energy.
- Axiom 2. The information axiom. Minimize the information content of the design. It is concerned with simplicity. In object-oriented system, to

minimize complexity use inheritance and the system's built in classes and add as little as possible to what already is there.

Real time example

https://www.youtube.com/watch?v=2vtT6TBnOAM&pbjreload=101

Designing Classes

OBJECTIVES :

students should be able to

- Designing classes
- Designing protocols and class visibility

COURSE OUTCOME

- Designing methods
- Develop packages

INTRODUCTION

- Most important activity in designing an application is coming up with a set of classes that work together to provide the needed functionality
- Use the OCL (Object Constraint Language- a specification language, provided by UML) to specify the properties of a system

Designing Classes

- Object-oriented design requires taking the object identified during object- oriented analysis and designing classes to represent them.
- As a class designer, we have to know the specifics of the class we are designing and also we should be aware of how that class interacts with other

  classes.

Object oriented design philosophy

- Here one has to think in terms of classes. As new facts are acquired, we relate them to existing structures in our environment (model).
- After enough new facts are acquired about a certain area, we create new structures to accommodate the greater level of detail in our knowledge.
- The important activity in designing an application is coming up with a set of classes that work together to provide the functionality we desire.
- If we design the classes with reusability in mind, we will gain a lot of productivity and reduce the time for developing new applications.

Designing Classes : the process

1. apply design axioms to design classes, their attributes, methods, associations, structures, & protocols

   1.1 refine and complete the static UML class diagram by adding details to that diagram

      1.1.1 refine attributes
      1.1.2 design methods and the protocols by utilizing a UML activity diagram to represent the method's algorithm
      1.1.3 refine the associations between classes (if required)
      1.1.4 refine the class hierarchy and design with inheritance (if required)

   1.2 Iterate and refine

Class visibility: Designing well-defined public, private and protected protocols

- In designing methods or attributes for classes,One is the protocol or interface to the class operations and its visibility and the other is how it is implemented.

- The class's protocol or the messages that a class understands, can be hidden from other objects (private protocol) or made available to other objects (public protocol).
- Public protocols define the functionality and external messages of an object.
- Private protocols define the implementation of an object.

Visibility

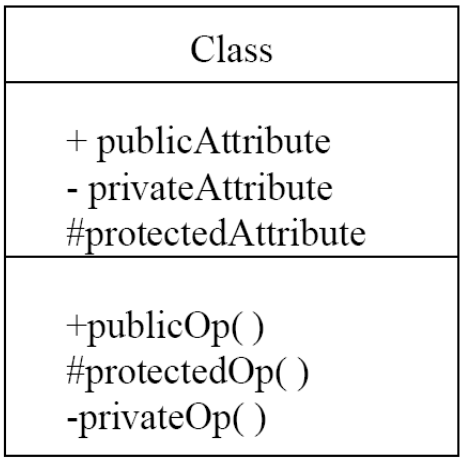
- A class might have a set of methods that it uses only internally, messages to itself. This private protocol of the class, includes messages that normally should not be sent from other objects. Here only the class itself can use the methods.
- The public protocol defines the stated behavior of the class as a citizen in a population and is important information for users as well as future descendants, so it is accessible to all classes. If the methods or attributes can be used by the class itself (or its subclasses) a protected protocol can be used. Here subclasses can used the method in addition to the class itself.
- The lack of well-designed protocol can manifest itself as encapsulation leakage. It happens when details about a class's internal implementation are disclosed through the interface.

Encapsulation leakage

- Encapsulation leakage is lack of a well-designed protocol
- The problem of encapsulation leakage occurs when details about a class's internal implementation are disclosed through the interface.
- As more internal details become visible, the flexibility to make changes in the future decreases.

Visibility types

- In UML, the following symbol are used to specify export control :
  - + public
  - # protected
  - - private



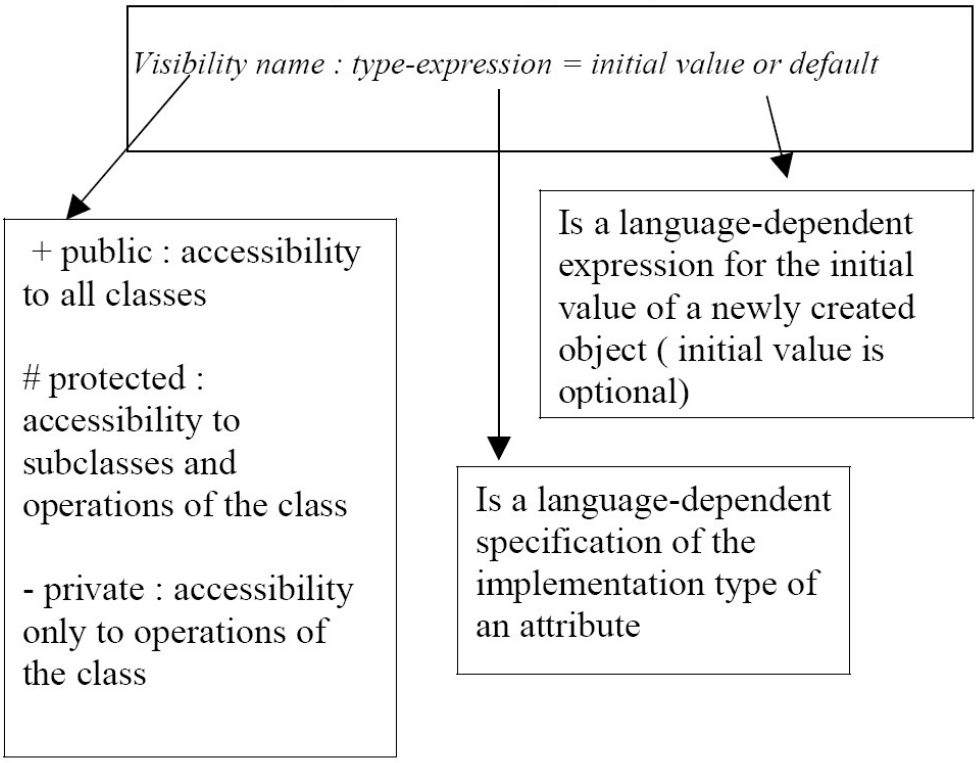
Attribute representations :

- During design, OCL (Object Constraint Language) can be used to define the class attributes

OCL

- The rules and semantics of the UML are expressed in English, in a form known as object constraint language.
- Object constraint language (OCL) is a specification language that uses simple logic for specifying the properties of a system.

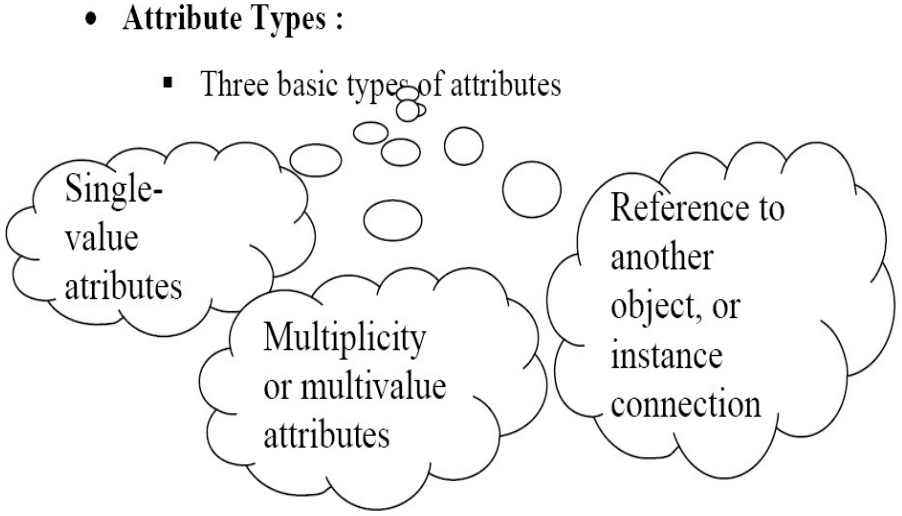
Attribute presentation suggested by UML :



Designing classes: Refining attributes

- Attributes identified in object-oriented analysis must be refined with an eye on implementation during this phase.
- In the analysis phase, the name of the attribute is enough.
- But in the design phase, detailed information must be added to the model.
- The 3 basic types of attributes are:

(1) Single-value attributes
(2) Multiplicity or multivalue attributes
(3) Reference to another object or instance connection



Attributes

- Attributes represent the state of an object.
- When the state of the object changes, these changes are reflected in the value of attributes.
- Single value attribute has only one value or state. (Eg). Name, address, salary
- Multiplicity or multivalue attribute can have a collection of many values at any time.
- (Eg) If we want to keep tact of the names of people who have called a customer support line for help.

Multi value attribute

- Instance connection attributes are required to provide the mapping needed by an object to fulfill its responsibilities.
- (E.g.) A person may have one or more bank accounts.

- A person has zero to many instance connections to Account(s).
- Similarly, an Account can be assigned to one or more person(s) (joint account).
- So an Account has zero to many instance connection to Person(s).

Designing methods and protocols

- A class can provide several types of methods:
- Constructor: Method that creates instances (objects) of the class
- Destructor: The method that destroys instances
- Conversion Method: The method that converts a value from one unit of measure to another.
- Copy Method: The method that copies the contents of one instance to another instance
- Attribute set: The method that sets the values of one or more attributes
- Attribute get: The method that returns the values of one or more attributes
- I/O methods: The methods that provide or receive data to or from a device
- Domain specific: The method specific to the application.

Access Layer - Object Storage

OBJECTIVES:

students should be able to

- Define and understand the persistent objects and transient objects
- Define and understand the object-relational systems

❒ **INTRODUCTION**

- DBMS = is a set of programs that enables the creation and maintenance of a collection of related data
- Fundamental purpose of a DBMS ➔ provide a reliable, persistent data storage facility and the mechanisms for efficient, convenient data access and retrieval
- In an OO system, it concerned with both persistent objects and transient objects
- Transient data ➔ data that will be erased from memory after they have been used
- Persistent data ➔ data that must be stored in a secondary data storage system, not just in computer memory, that must be stored after the program that creates or amends it stops running, and that usually must be available to other users

Access Layer: Object storage and object interoperability

- A Date Base Management System (DBMS) is a set of programs that enables the creation and maintenance (access, manipulate, protect and manage) of a collection of related data.
- The purpose of DBMS is to provide reliable, persistent data storage and mechanisms for efficient, convenient data access and retrieval.
- Persistence refers to the ability of some objects to outlive the programs that created them.
- Object lifetimes can be short for local objects (called transient objects) or long for objects stored indefinitely in a database (called persistent

objects).

Persistent stores

- Most object-oriented languages do not support serialization or object persistence, which is the process of writing or reading an object to and from a persistence storage medium, such as disk file.
- Unlike object oriented DBMS systems, the persistent object stores do not support query or interactive user interface facilities.
- Controlling concurrent access by users, providing ad-hoc query capability and allowing independent control over the physical location of data are not possible with persistent objects.

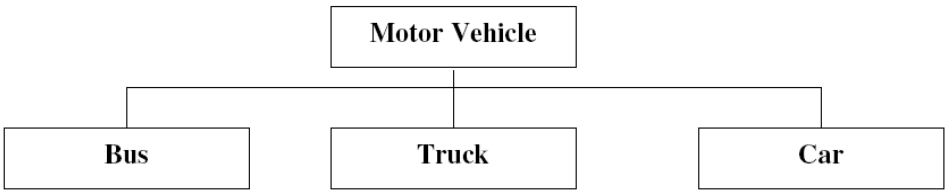
Object Store and Persistence

Each item of data will have a different lifetime. These lifetimes are categories into six, namely

1. Transient results to the evaluation of expressions.
2. Variable involved in procedure activation (parameters and variables with a localized scope)
3. Global variable and variables that are dynamically allocated
4. Data that exist between the executions of a program
5. Data that exist between the versions of a program.
6. Data that outlive a program.

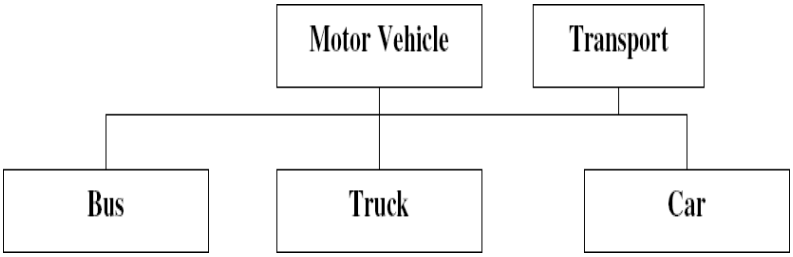
Database Models

- A database model is a collection of logical constructs used to represent the data structure and data relationships within the database. The conceptual model focuses on the logical nature of that data presentation. It is concerned with what is represented in the database and the implementation model is concerned with how it is represented.
- Hierarchical model
- This model represents data as a single rooted tree. Each node in the tree represents a data object and the connection represents a parent-child relationship.
- Network Model
- This is similar to a hierarchical database, with one difference. Here record can have more than one parent.



Network Model

- A network database model is similar to hierarchical model. Here in this model each parent can have any number of child nodes and each child node can have any number of parent nodes.



OODBMS

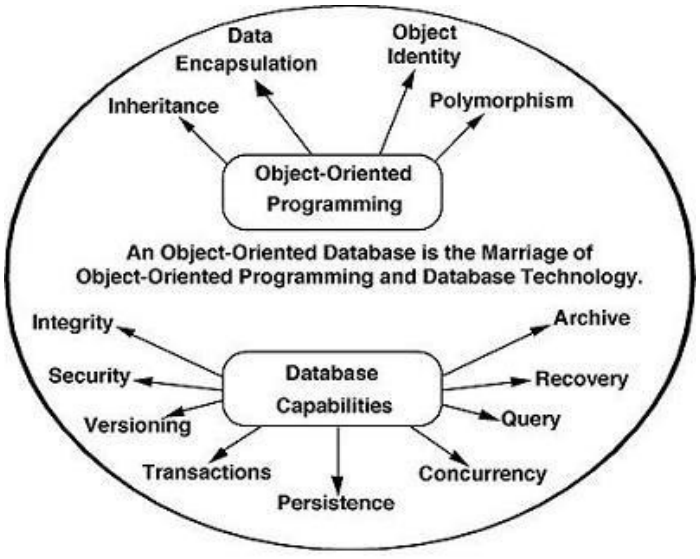


Figure 1. Makeup of an Object-Oriented Database

Rules to make object-oriented system

- The system must support complex objects.
- Object identity must be supported
- Objects must be encapsulated
- System must support types or classes
- System must support inheritance
- System must be extensible
- It must be persistent, able to remember an object state
- It must be able to manage large databases
- It must accept concurrent users
- Data query must be simple
- The system must support complex objects. System must provide simple atomic types of objects (integers, characters, etc) from which complex objects can be built by applying constructors to atomic objects.

- Object identity must be supported: A data object must have an identity and existence independent of its values.
- Objects must be encapsulated: An object must encapsulate both a program and its data.
- System must support types or classes: The system must support either the type concept or class concept
- System must support inheritance: Classes and types can participate in a class hierarchy. Inheritance factors out shared code and interfaces.
- System must avoid premature binding: This is also known as late binding or dynamic binding. It shows that the same method name can be used in different classes. The system must resolve conflicts in operation names at run time.
- System must be computationally complete: Any computable function should be expressible in DML of the system, allowing expression of any type of operation..

Object oriented databases versus Traditional Databases

The responsibility of an OODBMS includes definition of the object structures, object manipulation and recovery, which is the ability to maintain data integrity regardless of system, network or media failure.

The OODBMs like DBMSs must allow for sharing; secure, concurrent multiuser access; and efficient, reliable system performance.

The objects are an "active" component in an object-oriented database, in contrast to conventional database systems, where records play a passive role. Another feature of object-oriented database is inheritance. Relational databases do not explicitly provide inheritance of attributes and methods.

The objects are an active component in an object-oriented database, in contrast to conventional database systems, where records play a passive role.

The relational database systems do not explicitly provide inheritance of attributes and methods while object-oriented databases represent relationships explicitly (openly, clearly). (improvement in data access performance)

Object oriented databases also differ from the traditional relational databases in that they allow representation and storage of data in the form of objects. (each object has its own identity or object-ID

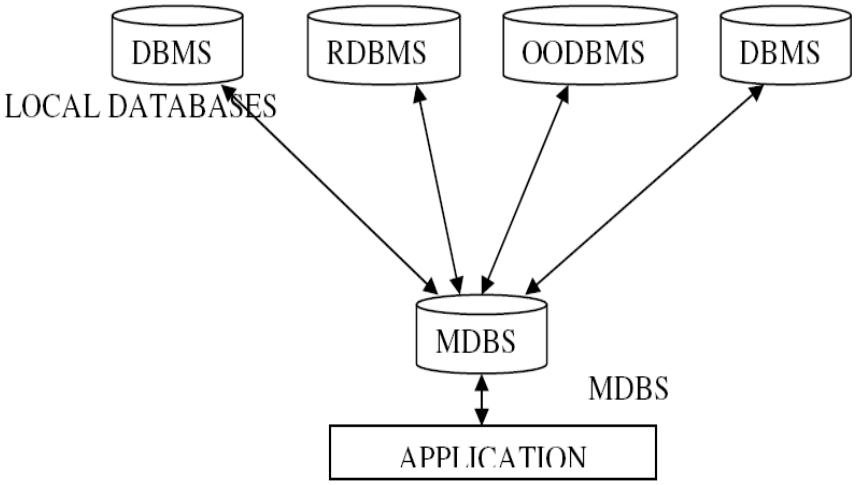
Object Identity

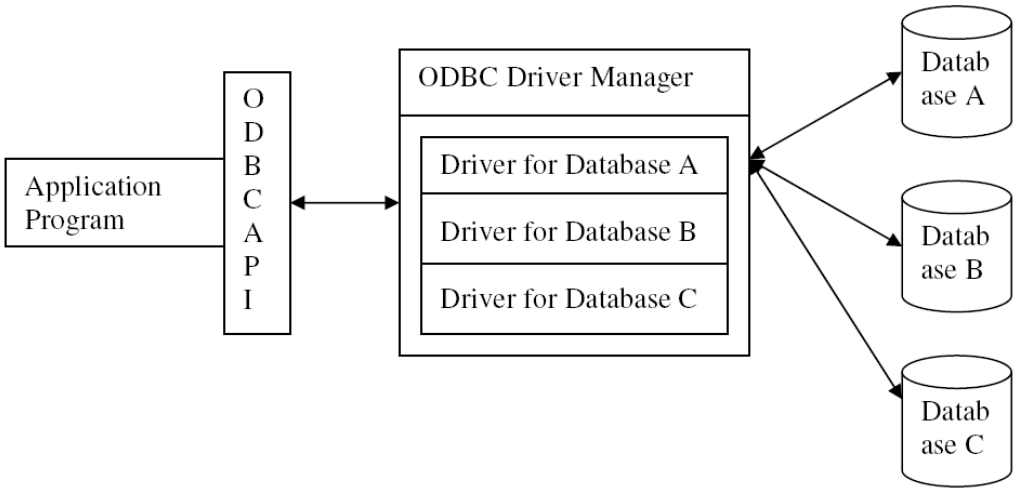
Object oriented databases allow representation and storage of data in the form of objects.

Each object has its own identity or object-ID (as opposed to the purely value- oriented approach of traditional databases).

The object identity is independent of the state of the object.

Multidatabase Systems




Application Program
O D B C A P I
ODBC Driver Manager
Driver for Database A
Driver for Database B
Driver for Database C
Datab ase A
Datab ase B
Datab ase C